dev-notes

RocFang

Published
with GitBook

目錄

RocFang的编程笔记

这本书最初只是作为测试gitbook的功能而用的,添加了几篇文章后,效果看起来不错。于是干脆用它来作为我的笔记本了,这个笔记没有一个完整的主题,基本就是个杂货铺,想到到哪写到哪。

本笔记的在线阅读地址为:dev-notes

本笔记主要书写是在gitbook上进行,但同时在github上有备份。github备份的地址为:dev-notes

网络编程

close与shutdown

shutdown的原型为：

```
#include <sys/socket.h>
int shutdown(int s, int how);
```

how可以为如下值：

1. SHUT_RD:调用后不能从该socket中读入数据。
2. SHUT_WR:调用后不能向该socket中写入数据。
3. SHUT_RDWR:综合1和2.即，既不能从该socket中读入数据，也不能向该socket中写入数据。

既然我们已经有了close,为什么还需要shutdown呢？主要有两点：

1. close本身并不会释放该socket占用的内存,而只是将该socket的引用计数减一。只有当该socket的引用计数减到0时,才会释放相关内存。 所以，调用close并不能保证该socket不可用了，而调用shutdown，可以直接关闭该连接。
2. shudown可以单方向的关闭一个socket连接，由参数how控制。

read/write与recv/send的区别

在功能上，read/write是recv/send的子集。read/wirte是更通用的文件描述符操作，而recv/send在socket领域则更“专业”一些。

当recv/send的flag参数设置为0时，则和read/write是一样的。

如果有如下几种需求，则read/write无法满足，必须使用recv/send:

1. 为接收和发送进行一些选项设置
2. 从多个客户端中接收报文
3. 发送带外数据（out-of-band data)

listen中的backlog参数的含义

listen的原型为:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

其中backlog有什么意义呢？

关于这个参数的很多说法都很模糊。

在《unix环境高级编程》第三版中是这么说的:

> The backlog argument provides a hint to the system regarding the number of outstanding connect requests that it should enqueue on behalf of the process. The actual value is determined by the system, but the upper limit is specified as SOMAXCONN in .

连"hint"这种词都用上了，说的是相当模糊，对吗？

我们再来看看《unix网络编程》第三版:

> The listen function is called only by a TCP server and it performs two actions:
>
> 1 . When a socket is created by the socket function, it is assumed to be an active socket, that is, a client socket that will issue a connect. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4), the call to listen moves the socket from the CLOSED state to the LISTEN state.
>
> 2 . The second argument to this function specifies the maximum number of connections the kernel should queue for this socket.
>
> This function is normally called after both the socket and bind functions and must be called before calling the accept function.
>
> To understand the backlog argument, we must realize that for a given listening socket, the kernel maintains two queues:
>
> 1 . An incomplete connection queue, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).

> 2 . A completed connection queue, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).
>
> When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved. Figure 4.8 depicts the packets exchanged during the connection establishment with these two queues.
>
> When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN (Section 2.6). This entry will remain on the incomplete queue until the third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or until the entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.) If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue. When the process calls accept, which we will describe in the next section, the first entry on the completed queue is returned to the process, or if the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.
>
> There are several points to consider regarding the handling of these two queues.
>
> 1.The backlog argument to the listen function has historically specified the maximum value for the sum of both queues.
>
> There has never been a formal definition of what the backlog means. The 4.2BSD man page says that it "defines the maximum length the queue of pending connections may grow to." Many man pages and even the POSIX specification copy this definition verbatim, but this definition does not say whether a pending connection is one in the SYN_RCVD state, one in the ESTABLISHED state that has not yet been accepted, or either. The historical definition in this bullet is the Berkeley implementation, dating back to 4.2BSD, and copied by many others.
>
> 2.Berkeley-derived implementations add a fudge factor to the backlog: It is multiplied by 1.5 (p. 257 of TCPv1 and p. 462 of TCPV2). For example, the commonly specified backlog of 5 really allows up to 8 queued entries on these systems, as we show in Figure 4.10.
>
> The reason for adding this fudge factor appears lost to history [Joy 1994]. But if we consider the backlog as specifying the maximum number of completed connections that the kernel will queue for a socket ([Borman 1997b], as discussed shortly), then the reason for the fudge factor is to take into account incomplete connections on the queue.

> 3.Do not specify a backlog of 0, as different implementations interpret this differently (Figure 4.10). If you do not want any clients connecting to your listening socket, close the listening socket.
>
> 4.Assuming the three-way handshake completes normally (i.e., no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT, whatever that value happens to be between a particular client and server. Section 14.4 of TCPv3 shows that for one Web server, the median RTT between many clients and the server was 187 ms. (The median is often used for this statistic, since a few large values can noticeably skew the mean.)
>
> 5.Historically, sample code always shows a backlog of 5, as that was the maximum value supported by 4.2BSD. This was adequate in the 1980s when busy servers would handle only a few hundred connections per day. But with the growth of the World Wide Web (WWW), where busy servers handle millions of connections per day, this small number is completely inadequate (pp. 187 192 of TCPv3). Busy HTTP servers must specify a much larger backlog, and newer kernels must support larger values.
>
> Many current systems allow the administrator to modify the maximum value for the backlog
>
> 6.A problem is: What value should the application specify for the backlog, since 5 is often inadequate? There is no easy answer to this. HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server. Another method is to assume some default but allow a command-line option or an environment variable to override the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error (p. 456 of TCPv2).
>
> 7.Manuals and books have historically said that the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to accept. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy Web servers have shown that this is false. The reason for specifying a large backlog is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.
>
> 8.If the queues are full when a client SYN arrives, TCP ignores the arriving SYN (pp. 930 931 of TCPv2); it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP immediately responded with an RST, the client's connect would return an error, forcing the application to handle this condition

> instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."
>
> Some implementations do send an RST when the queue is full. This behavior is ncorrect for the reasons stated above, and unless your client specifically needs to interact with such a server, it's best to ignore this possibility. Coding to handle this case reduces the robustness of the client and puts more load on the network in the normal RST case, where the port really has no server listening on it.
>
> 9.Data that arrives after the three-way handshake completes, but before the server calls accept, should be queued by the server TCP, up to the size of the connected socket's receive buffer.
>
> AIX and MacOS have the traditional Berkeley algorithm, and Solaris seems very close to that algorithm as well. FreeBSD just adds one to backlog.
>
> As we said, historically the backlog has specified the maximum value for the sum of both queues. During 1996, a new type of attack was launched on the Internet called SYN flooding [CERT 1996b]. The hacker writes a program to send SYNs at a high rate to the victim, filling the incomplete connection queue for one or more TCP ports. (We use the term hacker to mean the attacker, as described in [Cheswick, Bellovin, and Rubin 2003].) Additionally, the source IP address of each SYN is set to a random number (this is called IP spoofing) so that the server's SYN/ACK goes nowhere. This also prevents the server from knowing the real IP address of the hacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a denial of service to legitimate clients. There are two commonly used methods of handling these attacks, summarized in [Borman 1997b]. But what is most interesting in this note is revisiting what the listen backlog really means. It should specify the maximum number of completed connections for a given socket that the kernel will queue. The purpose of having a limit on these completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them (for whatever reason). If a system implements this interpretation, as does BSD/OS 3.0, then the application need not specify huge backlog values just because the server handles lots of client requests (e.g., a busy Web server) or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, scenarios do occur where the traditional value of 5 is inadequate.

长长的引用的《unix网络编程》中的原文。从中，我们知道，这个backlog参数的定义历史以来并不明确，但一种传统的定义是，backlog是已成功连接队列和未成功连接队列中元素之和。但各系统的实际做法还不一样，而且，Linux中是什么样子的，该书中并未提及，也可能跟上面的说法都不一样。

在上面的引文中，作者在提到sync攻击时，表达了这样一种观点：

> But what is most interesting in this note is revisiting what the listen backlog really means. It should specify the maximum number of completed connections for a given socket that the kernel will queue.

即，backlog应该代表建连成功的连接的个数，而不是2个队列元素个数的总和。

那么，Linux中是怎么做的呢，我们先看看Linux下listen的man手册:

> The backlog parameter defines the maximum length the queue of pending connections may grow to. If a connection request arrives with the queue full the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that retries succeed.
>
> The behaviour of the backlog parameter on TCP sockets changed with Linux 2.2. Now it specifies the queue length for completely established sockets waiting to be accepted, instead of the number of incomplete connection requests. The maximum length of the queue for incomplete sockets can be set using the tcp_max_syn_backlog sysctl. When syncookies are enabled there is no logical maximum length and this sysctl setting is ignored. See tcp(7) for more information.

即，Linux的Manual里说，在Linux2.2后，backlog就是已成功建立连接，等待被accept的连接个数。与上面《unix网络编程》引文中，作者的个人看法一致。

另外，可以参考这篇文章： How TCP backlog works in Linux

进程间传递文件描述符

首先，必须声明，“进程间传递文件描述符”这个说法是错误的。

> 在处理文件时，内核空间和用户空间使用的主要对象是不同的。对用户程序来说，一个文件由一 个文件描述符标识。该描述符是一个整数，在所有有关文件的操作中用作标识文件的参数。文件描述 符是在打开文件时由内核分配，只在一个进程内部有效。两个不同进程可以使用同样的文件描述符， 但二者并不指向同一个文件。基于同一个描述符来共享文件是不可能的。
>
> 《深入理解Linux内核架构》

这里说的“进程间传递文件描述符”是指，A进程打开文件fileA,获得文件描述符为fdA,现在A进程要通过某种方法，根据fdA,使得另一个进程B,获得一个新的文件描述符fdB,这个fdB在进程B中的作用，跟fdA在进程A中的作用一样。即在fdB上的操作,即是对fileA的操作。

这看似不可能的操作，是怎么进行的呢？

答案是使用匿名**Unix**域套接字，即**socketpair()**和**sendmsg/recvmsg**来实现。

关于**socketpair**

> UNIX domain sockets provide both stream and datagram interfaces. The UNIX domain datagram service is reliable, however. Messages are neither lost nor delivered out of order. UNIX domain sockets are like a cross between sockets and pipes. You can use the network-oriented socket interfaces with them, or you can use the socketpair function to create a pair of unnamed, connected, UNIX domain sockets.
>
> APUE 3rd edition,17.2

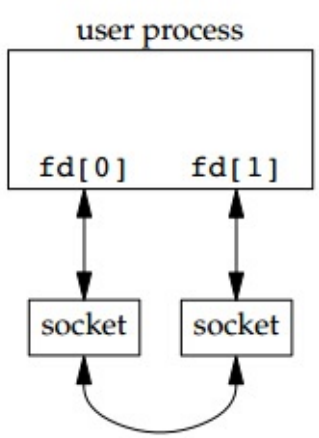
socketpair的原型为：

```
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int d, int type, int protocol, int sv[2]);
```

传入的参数sv为一个整型数组，有两个元素。当调用成功后，这个数组的两个元素即为2个文件描述符。

一对连接起来的Unix匿名域套接字就建立起来了，它们就像一个全双工的管道，每一端都既可读也可写。



即，往sv[0]写入的数据，可以通过sv[1]读出来，往sv[1]写入的数据，也可以通过sv[0]读出来。

关于sendmsg/recvmsg

通过socket发送数据，主要有三组系统调用，分别是

1. send/recv(与write/read类似，面向连接的)
2. sendto/recvfrom(sendto与send的差别在于，sendto可以面向无连接,recvfrom与recv的区别是,recvfrom可以获取sender方的地址)
3. sendmsg/recvmsg. 通过sendmsg,可以用msghdr参数，来指定多个缓冲区来发送数据，与writev系统调用类似。

sendmsg函数原型如下:

```
#include <sys/socket.h>
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

其中，根据POSIX.1 msghdr的定义至少应该包含下面几个成员：

```
struct msghdr {
    void *msg_name; /* optional address */
    socklen_t msg_namelen; /* address size in bytes */
    struct iovec *msg_iov; /* array of I/O buffers */
    int msg_iovlen; /* number of elements in array */
    void *msg_control; /* ancillary data */
    socklen_t msg_controllen; /* number of ancillary bytes */
    int msg_flags; /* flags for received message */
};
```

在Linux的manual page中，msghdr的定义为:

```
struct msghdr {
    void         *msg_name;       /* optional address */
    socklen_t     msg_namelen;    /* size of address */
    struct iovec *msg_iov;        /* scatter/gather array */
    size_t        msg_iovlen;     /* # elements in msg_iov */
    void         *msg_control;    /* ancillary data, see below */
    socklen_t     msg_controllen; /* ancillary data buffer len */
    int           msg_flags;      /* flags on received message */
};
```

查看Linux内核源代码(3.18.1)，可知msghdr的准确定义为：

```
struct msghdr {
    void        *msg_name;    /* ptr to socket address structure */
    int        msg_namelen;    /* size of socket address structure */
    struct iovec    *msg_iov;    /* scatter/gather array */
    __kernel_size_t    msg_iovlen;    /* # elements in msg_iov */
    void        *msg_control;    /* ancillary data */
    __kernel_size_t    msg_controllen;    /* ancillary data buffer length */
    unsigned int    msg_flags;    /* flags on received message */
};
```

可见，与Manual paga中的描述一致。

其中，前两个成员msg_name和msg_namelen是用来在发送datagram时，指定目的地址的。如果是面向连接的，这两个成员变量可以不用。

接下来的两个成员,msg_iov和msg_iovlen，则是用来指定发送缓冲区数组的。其中，msg_iovlen是iovec类型的元素的个数。每一个缓冲区的起始地址和大小由iovec类型自包含，iovec的定义为：

```
struct iovec {
    void *iov_base;   /* Starting address */
    size_t iov_len;   /* Number of bytes */
};
```

成员msg_flags用来描述接受到的消息的性质,由调用recvmsg时传入的flags参数设置。recvmsg的函数原型为:

```
#include <sys/socket.h>
ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

与sendmsg相对应，recvmsg用msghdr结构指定多个缓冲区来存放读取到的结果。flags参数用来修改recvmsg的默认行为。传入的flags参数在调用完recvmsg后，会被设置到msg所指向的msghdr类型的msg_flags变量中。flags可以为如下值:

Flag	Description	POSIX.1	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
MSG_CTRUNC	Control data was truncated.	•	•	•	•	•
MSG_EOR	End of record was received.	•	•	•	•	•
MSG_ERRQUEUE	Error information was received as ancillary data.			•		
MSG_OOB	Out-of-band data was received.	•	•	•	•	•
MSG_TRUNC	Normal data was truncated.	•	•	•	•	•

回来继续讲sendmsg和msghdr结构。

msghdr结构中剩下的两个成员,msg_control和msg_contorllen,是用来发送或接收控制信息的。其中,msg_control指向一个cmsghdr的结构体,msg_controllen成员是控制信息所占用的字节数。

注意,msg_controllen与前面的msg_iovlen不同,msg_iovlen是指的由成员msg_iov所指向的iovec型的数组的元素个数,而msg_controllen,则是所有控制信息所占用的总的字节数。

其实,msg_control也可能是个数组,但msg_controllen并不是该cmsghdr类型的数组的元素的个数。在Manual page中,关于msg_controllen有这么一段描述:

> To create ancillary data, first initialize the msg_controllen member of the msghdr with the length of the control message buffer. Use CMSG_FIRSTHDR() on the msghdr to get the first control message and CMSG_NEXTHDR to get all subsequent ones. In each control message, initialize cmsg_len (with CMSG_LEN), the other cmsghdr header fields, and the data portion using CMSG_DATA. Finally, the msg_controllen field of the msghdr should be set to the sum of the CMSG_SPACE() of the length of all control messages in the buffer.

在Linux 的Manual page(man cmsg)中,cmsghdr的定义为:

```
struct cmsghdr {
    socklen_t   cmsg_len;   /* data byte count, including header */
    int         cmsg_level; /* originating protocol */
    int         cmsg_type;  /* protocol-specific type */
    /* followed by  unsigned char   cmsg_data[]; */
};
```

注意,控制信息的数据部分,是直接存储在cmsg_type之后的。但中间可能有一些由于对齐产生的填充字节,由于这些填充数据的存在， 对于这些控制数据的访问,必须使用Linux提供的一些专用宏来完成。这些宏包括如下几个:

```
#include <sys/socket.h>

struct cmsghdr *CMSG_FIRSTHDR(struct msghdr *msgh);
struct cmsghdr *CMSG_NXTHDR(struct msghdr *msgh, struct cmsghdr *cmsg);
size_t CMSG_ALIGN(size_t length);
size_t CMSG_SPACE(size_t length);
size_t CMSG_LEN(size_t length);
unsigned char *CMSG_DATA(struct cmsghdr *cmsg);
```

其中:

CMSG_FIRSTHDR()返回msgh所指向的msghdr类型的缓冲区中的第一个cmsghdr结构体的指针。

CMSG_NXTHDR()返回传入的cmsghdr类型的指针的下一个cmsghdr结构体的指针。

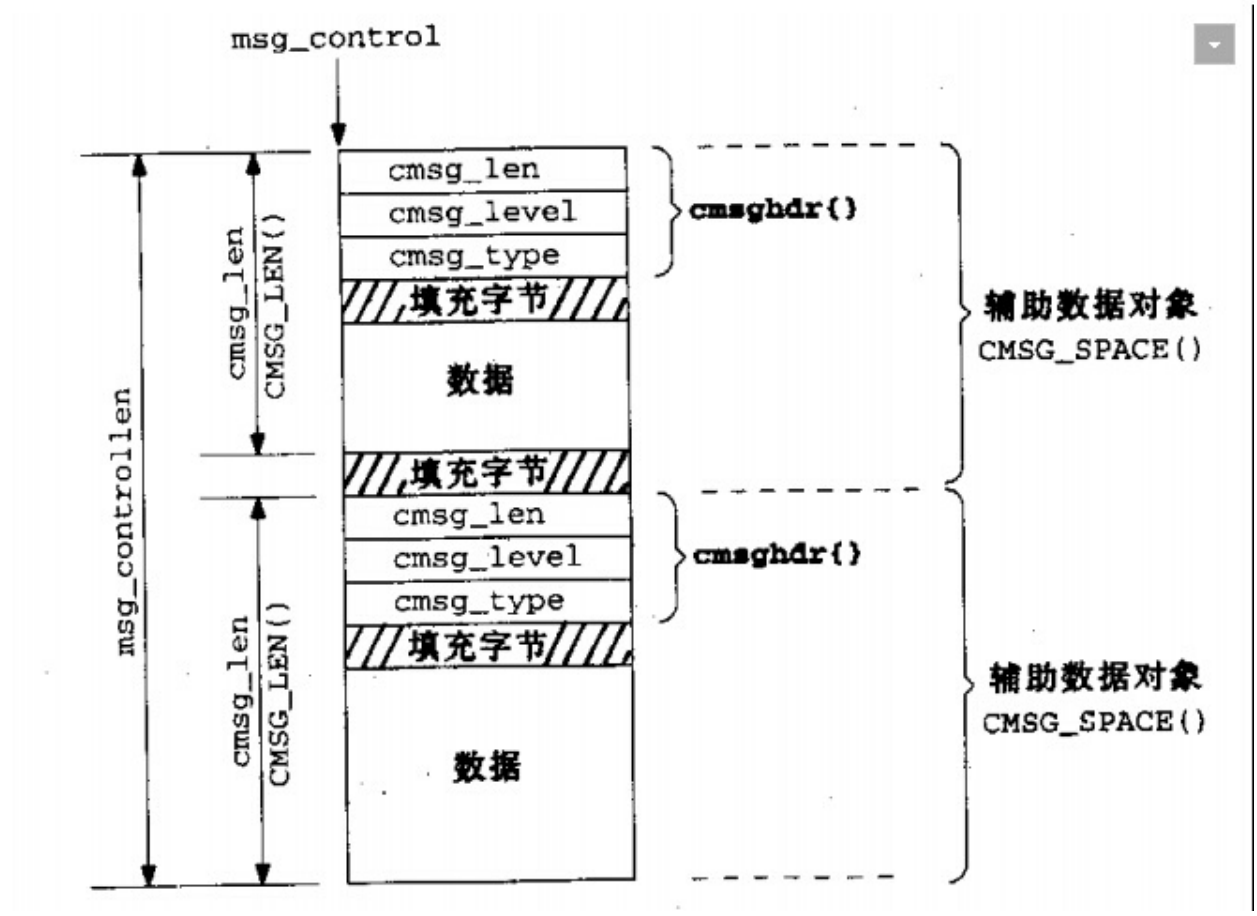
CMSG_ALIGN()根据传入的length大小,返回一个包含了添加对齐作用的填充数据后的大小。

CMSG_SPACE()中传入的参数length指的是一个控制信息元素(即一个cmsghdr结构体)后面数据部分的字节数,返回的是这个控制信息的总的字节数,即包含了头部(即cmsghdr各成员)、数据部分和填充数据的总和。

CMSG_DATA根据传入的cmsghdr指针参数,返回其后面数据部分的指针。

CMSG_LEN传入的参数是一个控制信息中的数据部分的大小,返回的是这个根据这个数据部分大小,需要配置的cmsghdr结构体中cmsg_len成员的值。这个大小将为对齐添加的填充数据也包含在内。

用一张图来表示这几个变量和宏的关系为:



如前所述,msghdr结构中,msg_controllen成员的大小为所有cmsghdr控制元素调用CMSG_SPACE()后相加的和。

讲了这么多msghdr,cmsghdr,还是没有讲到如何传递文件描述符。其实很简单,本来sendmsg是和send一样,是用来传送数据的,只不过其数据部分的buffer由参数msg_iov来指定,至此,其行为和send可以说是类似的。

但是sendmsg提供了可以传递控制信息的功能,我们要实现的传递描述符这一功能,就必须要用到这个控制信息。在msghdr变量的cmsghdr成员中,由控制头cmsg_level和cmsg_type来设置"传递文件描述符"这一属性,并将要传递的文件描述符作为数据部分,保存在cmsghdr变量的后面。这样就可以实现传递文件描述符这一功能,在此时,是不需要使用msg_iov来传递数据的。

具体的说,为msghdr的成员msg_control分配一个cmsghdr的空间,将该cmsghdr结构的cmsg_level设置为SOL_SOCKET,cmsg_type设置为SCM_RIGHTS,并将要传递的文件描述符作为数据部分,调用sendmsg即可。其中,SCM表示socket-level control message,SCM_RIGHTS表示我们要传递访问权限。

弄清楚了发送部分,文件描述符的接收部分就好说了。跟发送部分一样,为控制信息配置好属性,并在其后分配一个文件描述符的数据部分后，在成功调用recvmsg后,控制信息的数据部分就是在接收进程中的新的文件描述符了,接收进程可直接对该文件描述符进行操作。

Nginx中传递文件描述符的代码实现

关于如何在进程间传递文件描述符,我们已经理的差不多了。下面看看Nginx中是如何做的。

Nginx中发送文件描述符的相关代码为:

```
ngx_int_t
ngx_write_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size,
    ngx_log_t *log)
{
    ssize_t             n;
    ngx_err_t           err;
    struct iovec        iov[1];
    struct msghdr       msg;

#if (NGX_HAVE_MSGHDR_MSG_CONTROL)

    union {
        struct cmsghdr  cm;
        char            space[CMSG_SPACE(sizeof(int))];
    } cmsg;

    if (ch->fd == -1) {
        msg.msg_control = NULL;
        msg.msg_controllen = 0;

    } else {
        msg.msg_control = (caddr_t) &cmsg;
        msg.msg_controllen = sizeof(cmsg);

        ngx_memzero(&cmsg, sizeof(cmsg));

        cmsg.cm.cmsg_len = CMSG_LEN(sizeof(int));
        cmsg.cm.cmsg_level = SOL_SOCKET;
        cmsg.cm.cmsg_type = SCM_RIGHTS;

        /*
         * We have to use ngx_memcpy() instead of simple
         *   *(int *) CMSG_DATA(&cmsg.cm) = ch->fd;
         * because some gcc 4.4 with -O2/3/s optimization issues the warning:
         *   dereferencing type-punned pointer will break strict-aliasing rules
         *
         * Fortunately, gcc with -O1 compiles this ngx_memcpy()
         * in the same simple assignment as in the code above
         */

        ngx_memcpy(CMSG_DATA(&cmsg.cm), &ch->fd, sizeof(int));
    }

    msg.msg_flags = 0;
```

```
#else

    if (ch->fd == -1) {
        msg.msg_accrights = NULL;
        msg.msg_accrightslen = 0;

    } else {
        msg.msg_accrights = (caddr_t) &ch->fd;
        msg.msg_accrightslen = sizeof(int);
    }

#endif

    iov[0].iov_base = (char *) ch;
    iov[0].iov_len = size;

    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    n = sendmsg(s, &msg, 0);

    if (n == -1) {
        err = ngx_errno;
        if (err == NGX_EAGAIN) {
            return NGX_AGAIN;
        }

        ngx_log_error(NGX_LOG_ALERT, log, err, "sendmsg() failed");
        return NGX_ERROR;
    }

    return NGX_OK;
}
```

其中,参数s就是一个用socketpair创建的管道的一端,要传送的文件描述符位于参数ch所指向的结构体中。ch结构体本身,包含要传送的文件描述符和其他成员,则通过io_vec类型的成员msg_iov传送。

接收部分的代码为:

```
ngx_int_t
ngx_read_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size, ngx_log_t *log)
{
    ssize_t             n;
    ngx_err_t           err;
    struct iovec        iov[1];
    struct msghdr       msg;

#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
```

```
    union {
        struct cmsghdr  cm;
        char            space[CMSG_SPACE(sizeof(int))];
    } cmsg;
#else
    int                 fd;
#endif

    iov[0].iov_base = (char *) ch;
    iov[0].iov_len = size;

    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

#if (NGX_HAVE_MSGHDR_MSG_CONTROL)
    msg.msg_control = (caddr_t) &cmsg;
    msg.msg_controllen = sizeof(cmsg);
#else
    msg.msg_accrights = (caddr_t) &fd;
    msg.msg_accrightslen = sizeof(int);
#endif

    n = recvmsg(s, &msg, 0);

    if (n == -1) {
        err = ngx_errno;
        if (err == NGX_EAGAIN) {
            return NGX_AGAIN;
        }

        ngx_log_error(NGX_LOG_ALERT, log, err, "recvmsg() failed");
        return NGX_ERROR;
    }

    if (n == 0) {
        ngx_log_debug0(NGX_LOG_DEBUG_CORE, log, 0, "recvmsg() returned zero");
        return NGX_ERROR;
    }

    if ((size_t) n < sizeof(ngx_channel_t)) {
        ngx_log_error(NGX_LOG_ALERT, log, 0,
                      "recvmsg() returned not enough data: %z", n);
        return NGX_ERROR;
    }

#if (NGX_HAVE_MSGHDR_MSG_CONTROL)

    if (ch->command == NGX_CMD_OPEN_CHANNEL) {

        if (cmsg.cm.cmsg_len < (socklen_t) CMSG_LEN(sizeof(int))) {
            ngx_log_error(NGX_LOG_ALERT, log, 0,
```

```
                            "recvmsg() returned too small ancillary data");
                return NGX_ERROR;
            }

            if (cmsg.cm.cmsg_level != SOL_SOCKET || cmsg.cm.cmsg_type != SCM_RIGHTS)
            {
                ngx_log_error(NGX_LOG_ALERT, log, 0,
                              "recvmsg() returned invalid ancillary data "
                              "level %d or type %d",
                              cmsg.cm.cmsg_level, cmsg.cm.cmsg_type);
                return NGX_ERROR;
            }

            /* ch->fd = *(int *) CMSG_DATA(&cmsg.cm); */

            ngx_memcpy(&ch->fd, CMSG_DATA(&cmsg.cm), sizeof(int));
        }

        if (msg.msg_flags & (MSG_TRUNC|MSG_CTRUNC)) {
            ngx_log_error(NGX_LOG_ALERT, log, 0,
                          "recvmsg() truncated data");
        }

    #else

        if (ch->command == NGX_CMD_OPEN_CHANNEL) {
            if (msg.msg_accrightslen != sizeof(int)) {
                ngx_log_error(NGX_LOG_ALERT, log, 0,
                              "recvmsg() returned no ancillary data");
                return NGX_ERROR;
            }

            ch->fd = fd;
        }

    #endif

        return n;
    }
```

该代码配合发送部分的代码来读,意义很明确。只不过,在我们上面所讲的基础上,Nginx将ch变量作为发送和接收的数据(此数据指放在iovec缓冲区中的数据,而非控制信息中的数据部分),并用一个成员ch->command实现了一个简单的协议,使得这一对函数功能更通用。

计算机体系结构

大小端判断

定义：大端是高位（MSB）在地址靠前的部分，小端是低位(LSB)在地址靠前的部分。网络字节序是大端序，x86是小端序。

判断方法：

```
#include<stdio.h>

int main()
{
    int i = 0x11223344;
    char *p;

    p = (char *) &i;
    if (*p == 0x44)
    {
        printf("%s\n", "Little Endian!");
    }
    else
    {
        printf("%s\n", "Big Endian!");
    }
}
```

Nginx相关

Nginx中的几个基本模块和几个模块ctx类型

1.Nginx中的基本模块类型

Nginx中的模块对外看来主要包含如下几个类别：

1. Core模块
2. HTTP模块（标准HTTP模块和第三方HTTP模块）
3. Mail模块

其中，Core模块是最基本的模块。由于Nginx主要是一个HTTP服务器，所以默认编译时会把HTTP模块给编译进去，而不会编译Mail模块。如果要排除HTTP模块，则在编译时加入选项--without-http,同理，如果要包含Mail模块，则在编译时加入选项--with-mail.

2.Nginx的基本模块

我们先看Nginx有哪些基本模块，将HTTP模块排除在外。执行如下操作:

```
cd nginx-1.6.2
./configure --without-http
```

此时，会生成objs目录，打开objs/ngx_modules.c，可以看到其内容为:

```
#include <ngx_config.h>
#include <ngx_core.h>



extern ngx_module_t  ngx_core_module;
extern ngx_module_t  ngx_errlog_module;
extern ngx_module_t  ngx_conf_module;
extern ngx_module_t  ngx_events_module;
extern ngx_module_t  ngx_event_core_module;
extern ngx_module_t  ngx_epoll_module;

ngx_module_t *ngx_modules[] = {
    &ngx_core_module,
    &ngx_errlog_module,
    &ngx_conf_module,
    &ngx_events_module,
    &ngx_event_core_module,
    &ngx_epoll_module,
    NULL
};
```

其中，ngx_modules数组中包含的几个模块即为Nginx中的几个基本模块。

1. ngx_core_module
2. ngx_errlog_module
3. ngx_conf_module
4. ngx_events_module
5. ngx_event_core_module
6. ngx_epoll_module

3.Nginx模块的基本数据结构

每一个Nginx模块，都是一个ngx_module_t类型的变量，ngx_module_t类型定义如下:

```
struct ngx_module_s {
    ngx_uint_t            ctx_index;
    ngx_uint_t            index;

    ngx_uint_t            spare0;
    ngx_uint_t            spare1;
    ngx_uint_t            spare2;
    ngx_uint_t            spare3;

    ngx_uint_t            version;

    void                 *ctx;
    ngx_command_t        *commands;
    ngx_uint_t            type;

    ngx_int_t           (*init_master)(ngx_log_t *log);

    ngx_int_t           (*init_module)(ngx_cycle_t *cycle);

    ngx_int_t           (*init_process)(ngx_cycle_t *cycle);
    ngx_int_t           (*init_thread)(ngx_cycle_t *cycle);
    void                (*exit_thread)(ngx_cycle_t *cycle);
    void                (*exit_process)(ngx_cycle_t *cycle);

    void                (*exit_master)(ngx_cycle_t *cycle);

    uintptr_t             spare_hook0;
    uintptr_t             spare_hook1;
    uintptr_t             spare_hook2;
    uintptr_t             spare_hook3;
    uintptr_t             spare_hook4;
    uintptr_t             spare_hook5;
    uintptr_t             spare_hook6;
    uintptr_t             spare_hook7;
};
```

例如，http模块的定义为:

```
ngx_module_t  ngx_http_module = {
    NGX_MODULE_V1,
    &ngx_http_module_ctx,                  /* module context */
    ngx_http_commands,                     /* module directives */
    NGX_CORE_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    NULL,                                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

从上述结构体中，我们可以看到一个ctx成员。ctx成员是Nginx模块的重要组成部分。基本上，每一个Nginx模块（我们一般说一个Nginx模块，指的就是一个ngx_module_t类型的变量）都会定义一个ctx变量，之所以说基本上，是因为也有例外。例如ngx_conf_module的定义为:

```
ngx_module_t  ngx_conf_module = {
    NGX_MODULE_V1,
    NULL,                                  /* module context */
    ngx_conf_commands,                     /* module directives */
    NGX_CONF_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    ngx_conf_flush_files,                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

可以看到，其ctx成员为NULL。 在ngx_module_t的类型定义中，我们看到ctx是一个void*类型的指针，意味着其类型可能不是唯一的。 那么Nginx中有几种类型的ctx呢？

4.Nginx中的几个基本的ctx类型

Nginx中目前主要包含如下几个ctx类型：

1. ngx_core_module_t
2. ngx_event_module_t
3. ngx_http_module_t

4. ngx_mail_module_t
5. 其他

4.1 ngx_core_module_t

ngx_core_module_t的定义为:

```
typedef struct {
    ngx_str_t             name;
    void               *(*create_conf)(ngx_cycle_t *cycle);
    char               *(*init_conf)(ngx_cycle_t *cycle, void *conf);
} ngx_core_module_t;
```

Nginx-1.6.2中，ngx_core_module_t类型的模块（即ctx成员为ngx_core_module_t类型的ngx_module_t变量)包括：

1. ngx_core_module
2. ngx_events_module
3. ngx_openssl_module
4. ngx_google_perftools_module
5. ngx_http_module
6. ngx_errlog_module
7. ngx_mail_module
8. ngx_regex_module

这些模块的type成员均为常量NGX_CORE_MODULE，在遍历ngx_modules数组时，经常用该变量作为ngx_core_module_t类型模块的标记。

4.2 ngx_event_module_t

ngx_event_module_t的定义为:

```
typedef struct {
    ngx_str_t              *name;

    void                 *(*create_conf)(ngx_cycle_t *cycle);
    char                 *(*init_conf)(ngx_cycle_t *cycle, void *conf);

    ngx_event_actions_t     actions;
} ngx_event_module_t;

typedef struct {
    ngx_int_t  (*add)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t  (*del)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t  (*enable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t  (*disable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t  (*add_conn)(ngx_connection_t *c);
    ngx_int_t  (*del_conn)(ngx_connection_t *c, ngx_uint_t flags);

    ngx_int_t  (*process_changes)(ngx_cycle_t *cycle, ngx_uint_t nowait);
    ngx_int_t  (*process_events)(ngx_cycle_t *cycle, ngx_msec_t timer,
                   ngx_uint_t flags);

    ngx_int_t  (*init)(ngx_cycle_t *cycle, ngx_msec_t timer);
    void       (*done)(ngx_cycle_t *cycle);
} ngx_event_actions_t;
```

Nginx-1.6.2中，ngx_event_module_t类型的模块（即ctx成员为ngx_event_module_t类型的ngx_module_t变量)包括：

1. ngx_aio_module
2. ngx_devpoll_module
3. ngx_epoll_module
4. ngx_event_core_module
5. ngx_eventport_module
6. ngx_kqueue_module
7. ngx_poll_module
8. ngx_rtsig_module
9. ngx_select_module

一般来说，在Linux上，需要关注的是ngx_epoll_module和ngx_event_core_module.

这些模块的type成员均为常量NGX_EVENT_MODULE，在遍历ngx_modules数组时，经常用该变量作为ngx_event_module_t类型模块的标记。

4.3 ngx_http_module_t

ngx_http_module_t的定义为：

```
typedef struct {
    ngx_int_t   (*preconfiguration)(ngx_conf_t *cf);
    ngx_int_t   (*postconfiguration)(ngx_conf_t *cf);

    void       *(*create_main_conf)(ngx_conf_t *cf);
    char       *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void       *(*create_srv_conf)(ngx_conf_t *cf);
    char       *(*merge_srv_conf)(ngx_conf_t *cf, void *prev, void *conf);

    void       *(*create_loc_conf)(ngx_conf_t *cf);
    char       *(*merge_loc_conf)(ngx_conf_t *cf, void *prev, void *conf);
} ngx_http_module_t;
```

ngx_http_module_t类型的模块就太多了，我们通常说的Nginx模块编程，绝大部分即为此类。

Nginx-1.6.2中，原生的ngx_http_module_t类型的模块（即ctx为ngx_http_module_t类型的ngx_module_t变量)包括:

1. ngx_http_access_module
2. ngx_http_addition_filter_module
3. ngx_http_auth_basic_module
4. ngx_http_auth_request_module
5. ngx_http_autoindex_module
6. ngx_http_browser_module
7. ngx_http_charset_filter_module
8. ngx_http_chunked_filter_module
9. ngx_http_copy_filter_module
10. ngx_http_core_module
11. ngx_http_dav_module
12. ngx_http_degradation_module
13. ngx_http_empty_gif_module
14. ngx_http_fastcgi_module
15. ngx_http_flv_module
16. ngx_http_geoip_module
17. ngx_http_geo_module
18. ngx_http_gunzip_filter_module
19. ngx_http_gzip_filter_module
20. ngx_http_gzip_static_module
21. ngx_http_headers_filter_module
22. ngx_http_header_filter_module

23. ngx_http_image_filter_module
24. ngx_http_index_module
25. ngx_http_limit_conn_module
26. ngx_http_limit_req_module
27. ngx_http_log_module
28. ngx_http_map_module
29. ngx_http_memcached_module
30. ngx_http_mp4_module
31. ngx_http_not_modified_filter_module
32. ngx_http_perl_module
33. ngx_http_postpone_filter_module
34. ngx_http_proxy_module
35. ngx_http_random_index_module
36. ngx_http_range_header_filter_module
37. ngx_http_range_body_filter_module
38. ngx_http_realip_module
39. ngx_http_referer_module
40. ngx_http_rewrite_module
41. ngx_http_scgi_module
42. ngx_http_secure_link_module
43. ngx_http_spdy_filter_module
44. ngx_http_spdy_module
45. ngx_http_split_clients_module
46. ngx_http_ssi_filter_module
47. ngx_http_ssl_module
48. ngx_http_static_module
49. ngx_http_stub_status_module
50. ngx_http_sub_filter_module
51. ngx_http_upstream_module
52. ngx_http_upstream_ip_hash_module
53. ngx_http_upstream_keepalive_module
54. ngx_http_upstream_least_conn_module
55. ngx_http_userid_filter_module
56. ngx_http_uwsgi_module
57. ngx_http_write_filter_module
58. ngx_http_xslt_filter_module

这些模块的type成员均为常量NGX_HTTP_MODULE，在遍历ngx_modules数组时，经常用该变量作为ngx_http_module_t类型模块的标记。

4.4 ngx_mail_module_t

ngx_mail_module_t的定义为:

```
typedef struct {
    ngx_mail_protocol_t        *protocol;

    void                       *(*create_main_conf)(ngx_conf_t *cf);
    char                       *(*init_main_conf)(ngx_conf_t *cf, void *conf);

    void                       *(*create_srv_conf)(ngx_conf_t *cf);
    char                       *(*merge_srv_conf)(ngx_conf_t *cf, void *prev,
                                       void *conf);
} ngx_mail_module_t;
```

Nginx-1.6.2中，ngx_mail_module_t类型的模块（即ctx成员为ngx_mail_module_t类型的ngx_module_t变量)包括：

1. ngx_mail_auth_http_module
2. ngx_mail_core_module
3. ngx_mail_imap_module
4. ngx_mail_pop3_module
5. ngx_mail_proxy_module
6. ngx_mail_smtp_module
7. ngx_mail_ssl_module

这些模块的type成员均为常量NGX_MAIL_MODULE，在遍历ngx_modules数组时，经常用该变量作为ngx_mail_module_t类型模块的标记。

4.5其他模块

这里，主要是ngx_conf_module,其定义为:

```
ngx_module_t  ngx_conf_module = {
    NGX_MODULE_V1,
    NULL,                                  /* module context */
    ngx_conf_commands,                     /* module directives */
    NGX_CONF_MODULE,                       /* module type */
    NULL,                                  /* init master */
    NULL,                                  /* init module */
    NULL,                                  /* init process */
    NULL,                                  /* init thread */
    NULL,                                  /* exit thread */
    ngx_conf_flush_files,                  /* exit process */
    NULL,                                  /* exit master */
    NGX_MODULE_V1_PADDING
};
```

该模块有如下两点值得注意:

1. 如前所述，其ctx成员为NULL.
2. 其type为NGX_CONF_MODULE

Nginx中的ngx_modules数组

1.ngx_modules数组的产生

ngx_modules数组是在执行configure脚本后自动生成的，在objs/ngx_modules.c文件中。该数组即当前编译版本中的所有Nginx模块。

做如下操作，可以看到ngx_moduels数组的一般形式：

1.1 最简单的Nginx框架，不包含任何HTTP模块

```
cd nginx-1.6.2
./configure --without-http
```

此时，objs/ngx_modules.c的内容为:

```
#include <ngx_config.h>
#include <ngx_core.h>



extern ngx_module_t  ngx_core_module;
extern ngx_module_t  ngx_errlog_module;
extern ngx_module_t  ngx_conf_module;
extern ngx_module_t  ngx_events_module;
extern ngx_module_t  ngx_event_core_module;
extern ngx_module_t  ngx_epoll_module;

ngx_module_t *ngx_modules[] = {
    &ngx_core_module,
    &ngx_errlog_module,
    &ngx_conf_module,
    &ngx_events_module,
    &ngx_event_core_module,
    &ngx_epoll_module,
    NULL
};
```

1.2 默认的ngx_modules数组:

```
make clean
cd nginx-1.6.2
./configure
```

此时，objs/ngx_modules.c的内容为：

```
#include <ngx_config.h>
#include <ngx_core.h>



extern ngx_module_t  ngx_core_module;
extern ngx_module_t  ngx_errlog_module;
extern ngx_module_t  ngx_conf_module;
extern ngx_module_t  ngx_events_module;
extern ngx_module_t  ngx_event_core_module;
extern ngx_module_t  ngx_epoll_module;
extern ngx_module_t  ngx_regex_module;
extern ngx_module_t  ngx_http_module;
extern ngx_module_t  ngx_http_core_module;
extern ngx_module_t  ngx_http_log_module;
extern ngx_module_t  ngx_http_upstream_module;
extern ngx_module_t  ngx_http_static_module;
extern ngx_module_t  ngx_http_autoindex_module;
extern ngx_module_t  ngx_http_index_module;
extern ngx_module_t  ngx_http_auth_basic_module;
extern ngx_module_t  ngx_http_access_module;
extern ngx_module_t  ngx_http_limit_conn_module;
extern ngx_module_t  ngx_http_limit_req_module;
extern ngx_module_t  ngx_http_geo_module;
extern ngx_module_t  ngx_http_map_module;
extern ngx_module_t  ngx_http_split_clients_module;
extern ngx_module_t  ngx_http_referer_module;
extern ngx_module_t  ngx_http_rewrite_module;
extern ngx_module_t  ngx_http_proxy_module;
extern ngx_module_t  ngx_http_fastcgi_module;
extern ngx_module_t  ngx_http_uwsgi_module;
extern ngx_module_t  ngx_http_scgi_module;
extern ngx_module_t  ngx_http_memcached_module;
extern ngx_module_t  ngx_http_empty_gif_module;
extern ngx_module_t  ngx_http_browser_module;
extern ngx_module_t  ngx_http_upstream_ip_hash_module;
extern ngx_module_t  ngx_http_upstream_least_conn_module;
extern ngx_module_t  ngx_http_upstream_keepalive_module;
extern ngx_module_t  ngx_http_write_filter_module;
extern ngx_module_t  ngx_http_header_filter_module;
extern ngx_module_t  ngx_http_chunked_filter_module;
extern ngx_module_t  ngx_http_range_header_filter_module;
extern ngx_module_t  ngx_http_gzip_filter_module;
extern ngx_module_t  ngx_http_postpone_filter_module;
extern ngx_module_t  ngx_http_ssi_filter_module;
```

```
extern ngx_module_t  ngx_http_charset_filter_module;
extern ngx_module_t  ngx_http_userid_filter_module;
extern ngx_module_t  ngx_http_headers_filter_module;
extern ngx_module_t  ngx_http_copy_filter_module;
extern ngx_module_t  ngx_http_range_body_filter_module;
extern ngx_module_t  ngx_http_not_modified_filter_module;

ngx_module_t *ngx_modules[] = {
    &ngx_core_module,
    &ngx_errlog_module,
    &ngx_conf_module,
    &ngx_events_module,
    &ngx_event_core_module,
    &ngx_epoll_module,
    &ngx_regex_module,
    &ngx_http_module,
    &ngx_http_core_module,
    &ngx_http_log_module,
    &ngx_http_upstream_module,
    &ngx_http_static_module,
    &ngx_http_autoindex_module,
    &ngx_http_index_module,
    &ngx_http_auth_basic_module,
    &ngx_http_access_module,
    &ngx_http_limit_conn_module,
    &ngx_http_limit_req_module,
    &ngx_http_geo_module,
    &ngx_http_map_module,
    &ngx_http_split_clients_module,
    &ngx_http_referer_module,
    &ngx_http_rewrite_module,
    &ngx_http_proxy_module,
    &ngx_http_fastcgi_module,
    &ngx_http_uwsgi_module,
    &ngx_http_scgi_module,
    &ngx_http_memcached_module,
    &ngx_http_empty_gif_module,
    &ngx_http_browser_module,
    &ngx_http_upstream_ip_hash_module,
    &ngx_http_upstream_least_conn_module,
    &ngx_http_upstream_keepalive_module,
    &ngx_http_write_filter_module,
    &ngx_http_header_filter_module,
    &ngx_http_chunked_filter_module,
    &ngx_http_range_header_filter_module,
    &ngx_http_gzip_filter_module,
    &ngx_http_postpone_filter_module,
    &ngx_http_ssi_filter_module,
    &ngx_http_charset_filter_module,
    &ngx_http_userid_filter_module,
    &ngx_http_headers_filter_module,
    &ngx_http_copy_filter_module,
    &ngx_http_range_body_filter_module,
```

```
    &ngx_http_not_modified_filter_module,
    NULL
};
```

2.直接使用ngx_modules的场景

如上所述，ngx_modules数组代表了当前Nginx中的所有模块，由于每一个Nginx模块，即ngx_module_t类型的变量，其ctx变量一般为如下几个类型之一:

1. ngx_core_module_t
2. ngx_event_module_t
3. ngx_http_module_t
4. ngx_mail_module_t

所以，可以对所有的Nginx模块按其ctx类型进行分类，当然并不是所有的nginx模块都有ctx成员，例如ngx_conf_module模块的ctx成员为NULL,但可以认为绝大部分模块都属于上述四类模块之一。

每一类模块下的所有模块，由于其ctx结构是一样的，因而在程序执行逻辑上会有共同点。具体来说，我们可以遍历ngx_modules数组成员，判断其属于上述四类的哪一类模块，再调用其对应的ctx成员中的函数指针，这样就会屏蔽掉具体的模块名称，抽象到框架的层面上。

那么，如何判断一个模块属于上述四类模块中的哪一类呢？这就是模块变量的type成员的作用了。

1. 所有的ngx_core_module_t类型的模块，其type成员为NGX_CORE_MODULE
2. 所有的ngx_event_module_t类型的模块，其type成员为NGX_EVENT_MODULE
3. 所有的ngx_http_module_t类型的模块，其type成员为NGX_HTTP_MODULE
4. 所有的ngx_mail_module_t类型的模块，其type成员为NGX_MAIL_MODULE

所以,在遍历ngx_modules数组时，即可根据每一个数组成员的type成员，来判断该模块属于哪种类型的模块，进而执行该模块对应的钩子函数。

下面，我们以nginx-1.6.2为例，看看程序在哪些地方对nginx_modules数组进行了遍历。

2.1 统计模块数量

```
ngx_max_module = 0;
    for (i = 0; ngx_modules[i]; i++) {
        ngx_modules[i]->index = ngx_max_module++;
    }
```

上述代码在nginx.c的main函数中，可以看到，其作用为:

1. 统计系统中到底有多少个模块，记录在全局变量ngx_max_module中。
2. 将每个模块在ngx_modules数组中的序号，记录在模块的index变量中。

2.2 用来解析每个模块的配置

具体见函数ngx_conf_handler。

2.3 执行每个模块的**ctx**中的钩子函数

例如,ngx_init_cycle函数中:

```
for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_CORE_MODULE) {
            continue;
        }

        module = ngx_modules[i]->ctx;

        if (module->create_conf) {
            rv = module->create_conf(cycle);
            if (rv == NULL) {
                ngx_destroy_pool(pool);
                return NULL;
            }
            cycle->conf_ctx[ngx_modules[i]->index] = rv;
        }
    }
```

以上代码对所有的ngx_core_module_t类型的模块，调用其ctx成员中的create_conf钩子函数。

2.4 同样在在**ngx_init_cycle**函数：

```
for (i = 0; ngx_modules[i]; i++) {
      if (ngx_modules[i]->type != NGX_CORE_MODULE) {
          continue;
      }

      module = ngx_modules[i]->ctx;

      if (module->init_conf) {
          if (module->init_conf(cycle, cycle->conf_ctx[ngx_modules[i]->index])
              == NGX_CONF_ERROR)
          {
              environ = senv;
              ngx_destroy_cycle_pools(&conf);
              return NULL;
          }
      }
  }
```

以上代码对所有的ngx_core_module_t类型的模块，调用其ctx成员的init_conf钩子函数。

2.5 继续看ngx_init_cycle函数:

```
 for (i = 0; ngx_modules[i]; i++) {
      if (ngx_modules[i]->init_module) {
          if (ngx_modules[i]->init_module(cycle) != NGX_OK) {
              /* fatal */
              exit(1);
          }
      }
  }
```

以上代码调用所有的nginx模块（ngx_module_t变量）的init_module钩子函数（如果不为空的话）。

2.6 在ngx_event_process_init函数中:

```
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
            continue;
        }

        if (ngx_modules[m]->ctx_index != ecf->use) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->actions.init(cycle, ngx_timer_resolution) != NGX_OK) {
            /* fatal */
            exit(2);
        }

        break;
    }
```

调用event模块的ctx成员中的actions.init钩子函数，对于epoll,则是ngx_epoll_init函数。

2.7 在ngx_events_block函数中:

```
ngx_event_max_module = 0;
for (i = 0; ngx_modules[i]; i++) {
      if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
          continue;
      }

      ngx_modules[i]->ctx_index = ngx_event_max_module++;
  }
```

上面的代码，对于所有的NGX_EVENT_MODULE类型的模块，初始化其ctx_index变量。主要有两点:

1. ngx_event_max_module代表了ngx_event_module_t类型模块的个数。而ngx_max_module则代表了所有模块的个数。
2. ctx_index成员表示该模块在该类模块（本例中为ngx_event_module_t类型的模块)中的序号。

2.8 在ngx_events_block函数中:

```
for (i = 0; ngx_modules[i]; i++) {
       if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
           continue;
       }

       m = ngx_modules[i]->ctx;

       if (m->create_conf) {
           (*ctx)[ngx_modules[i]->ctx_index] = m->create_conf(cf->cycle);
           if ((*ctx)[ngx_modules[i]->ctx_index] == NULL) {
               return NGX_CONF_ERROR;
           }
       }
   }
```

上面的代码，对于所有的NGX_EVENT_MODULE类型的模块,调用其ctx成员中的create_conf钩子函数。

2.9 在ngx_events_block函数中:

```
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->type != NGX_EVENT_MODULE) {
            continue;
        }

        m = ngx_modules[i]->ctx;

        if (m->init_conf) {
            rv = m->init_conf(cf->cycle, (*ctx)[ngx_modules[i]->ctx_index]);
            if (rv != NGX_CONF_OK) {
                return rv;
            }
        }
    }
```

上面的代码，对于所有的NGX_EVENT_MODULE类型的模块,调用其ctx成员中的init_conf钩子函数。

2.10 在ngx_event_use函数中:

```
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_EVENT_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;
        if (module->name->len == value[1].len) {
            if (ngx_strcmp(module->name->data, value[1].data) == 0) {
                ecf->use = ngx_modules[m]->ctx_index;
                ecf->name = module->name->data;

                if (ngx_process == NGX_PROCESS_SINGLE
                    && old_ecf
                    && old_ecf->use != ecf->use)
                {
                    ngx_conf_log_error(NGX_LOG_EMERG, cf, 0,
                               "when the server runs without a master process "
                               "the \"%V\" event type must be the same as "
                               "in previous configuration - \"%s\" "
                               "and it cannot be changed on the fly, "
                               "to change it you need to stop server "
                               "and start it again",
                               &value[1], old_ecf->name);

                    return NGX_CONF_ERROR;
                }

                return NGX_CONF_OK;
            }
        }
    }
```

ngx_event_use是ngx_event_core_module模块的指令"use"的解析函数。

2.11 在函数ngx_http_block中:

```
    ngx_http_max_module = 0;
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        ngx_modules[m]->ctx_index = ngx_http_max_module++;
    }
```

上面的代码主要做了两件事情:

1. 遍历所有的ngx_http_module_t类型的模块，统计这种模块的个数，存在变量

ngx_http_max_module中。

2. 初始化每个ngx_http_module_t类型的模块的ctx_index变量，该值的涵义是每个ngx_http_module_t模块在所有该类模块中的序号。

2.12 在函数ngx_http_block中:

```
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;
        mi = ngx_modules[m]->ctx_index;

        if (module->create_main_conf) {
            ctx->main_conf[mi] = module->create_main_conf(cf);
            if (ctx->main_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }

        if (module->create_srv_conf) {
            ctx->srv_conf[mi] = module->create_srv_conf(cf);
            if (ctx->srv_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }

        if (module->create_loc_conf) {
            ctx->loc_conf[mi] = module->create_loc_conf(cf);
            if (ctx->loc_conf[mi] == NULL) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

上面的代码分别调用所有ngx_http_module_t类型模块的create_main_conf、create_srv_conf和create_loc_conf钩子函数。

2.13 同样在ngx_http_block函数中:

```
for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->preconfiguration) {
            if (module->preconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

调用所有ngx_http_module_t类型模块的preconfiguration钩子函数。

2.14 同样在ngx_http_block函数中:

```
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;
        mi = ngx_modules[m]->ctx_index;

        /* init http{} main_conf's */

        if (module->init_main_conf) {
            rv = module->init_main_conf(cf, ctx->main_conf[mi]);
            if (rv != NGX_CONF_OK) {
                goto failed;
            }
        }

        rv = ngx_http_merge_servers(cf, cmcf, module, mi);
        if (rv != NGX_CONF_OK) {
            goto failed;
        }
    }
```

上面的代码首先调用所有ngx_http_module_t类型模块的init_main_conf钩子函数，然后对main, server, location等配置进行了合并。

2.15 仍然看ngx_http_block函数:

```
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_HTTP_MODULE) {
            continue;
        }

        module = ngx_modules[m]->ctx;

        if (module->postconfiguration) {
            if (module->postconfiguration(cf) != NGX_OK) {
                return NGX_CONF_ERROR;
            }
        }
    }
```

很明显，上面的代码调用了所有ngx_http_module_t类型的模块的postconfiguration钩子函数。

2.16 在函数ngx_mail_block中:

```
    ngx_mail_max_module = 0;
    for (m = 0; ngx_modules[m]; m++) {
        if (ngx_modules[m]->type != NGX_MAIL_MODULE) {
            continue;
        }

        ngx_modules[m]->ctx_index = ngx_mail_max_module++;
    }
```

不难看出，该段代码主要完成两点功能：

1. 统计所有的ngx_mail_module_t类型的模块，存入变量ngx_mail_max_module中。
2. 初始化每个ngx_mail_module_t类型模块的ctx_index变量，该变量表示每个该类变量在所有ngx_mail_module_t变量中的序号。

mail与http类似，还有许多关于ngx_mail_module_t的ctx变量的钩子函数的调用，这里略过。

2.17 在函数ngx_single_process_cycle中:

```
for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->init_process) {
            if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) {
                /* fatal */
                exit(2);
            }
        }
    }
```

以上代码调用所有模块的init_process钩子函数。

2.18 在函数ngx_master_process_exit中:

```
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->exit_master) {
            ngx_modules[i]->exit_master(cycle);
        }
    }
```

以上代码调用所有模块的exit_process钩子函数。

2.19 在函数ngx_worker_process_init函数中:

```
   for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->init_process) {
            if (ngx_modules[i]->init_process(cycle) == NGX_ERROR) {
                /* fatal */
                exit(2);
            }
        }
    }
```

上面的代码调用所有模块的init_process钩子函数.

2.20 在函数ngx_worker_process_exit中:

```
    for (i = 0; ngx_modules[i]; i++) {
        if (ngx_modules[i]->exit_process) {
            ngx_modules[i]->exit_process(cycle);
        }
    }
```

上面的代码调用了所有模块的exit_process钩子函数。

Nginx中的进程间通信

我们知道,Linux提供了多种进程间传递消息的方式，比如共享内存、套接字、管道、消息队列、信号等，每种方式都各有特点，各有优缺点。其中Nginx主要使用了其中的三种方式:

- 套接字（匿名套接字对）
- 共享内存
- 信号

本文主要结合代码讲一下前两种方式，匿名套接字对和共享内存在Nginx中的使用。

1. Nginx中的channel通信机制

1.1概述

首先简单的说一下Nginx中channel通信的机制。

Nginx中的channel通信，本质上是多个进程之间,利用匿名套接字(socketpair)对来进行通信。

我们知道,socketpair可以创建出一对套接字,在这两个套接字的任何一个上面进行写操作,在另一个套接字上就可以相应的进行读操作,而且这个管道是全双工的。

那么,当父进程在调用了socketpair创建出一对匿名套接字对(A1,B1)后,fork出一个子进程,那么此时子进程也继承了这一对套接字对(A2,B2)。在这个基础上,父子进程即可进行通信了。例如,父进程对A1进行写操作,子进程可通过B2进行相应的读操作;子进程对B2进行写操作,父进程可以通过A1来进行相应的读操作等等。

我们假设,父进程依次fork了N个子进程,在每次fork之前,均如前所述调用了socketpair建立起一个匿名套接字对,这样,父进程与各个子进程之间即可通过各自的套接字对来进行通信。

但是各子进程之间能否使用匿名套接字对来进行通信呢？

我们假设父进程A中,它与子进程B之间的匿名套接字对为AB[2],它与子进程C之间的匿名套接字对为AC[2]。且进程B在进程C之前被fork出来。

对进程B而言，当它被fork出来后，它就继承了父进程创建的套接字对,命名为BA[2],这样父进程通过操作AB[2]，子进程B通过操作BA[2],即可实现父子进程之间的通信。

对进程C而言,当它被fork出来后，他就继承了父进程穿件的套接字对，命名为CA[2],这样父进程通过操作AC[2]，子进程C通过操作CA[2]，即可实现父子进程之间的通信。

但B和C有一点不同。由于B进程在C之前被fork，B进程无法从父进程中继承到父进程与C进程之间的匿名套接字对,而C进程在后面被fork出来，它却从父进程处继承到了父进程与子进程B之间的匿名套接字对。

这样，之后被fork出来的进程C,可以通过它从父进程那里继承到的与B进程相关联的匿名套接字对来向进程B发送消息，但进程B却无法向进程C发送消息。

当子进程数量比较多时，就会造成这样的情况：即后面的进程拥有前面每一个子进程的一个匿名套接字，但前面的进程则没有后面任何一个子进程的匿名套接字。

那么这个问题该如何解决呢？这就涉及到进程间传递文件描述符这个话题了。可以参考这里: 进程之间传递文件描述符。一个子进程被fork出来后，它可以依次向它之前被fork出来的所有子进程传递自己的描述符（匿名套接字对中的一个)。

通过这种机制,子进程之间也可以进行通信了。

Nginx中也就是这么做的。

1.2 Nginx中的具体实现

在ngx_process.c中，定义了一个全局的数组ngx_processes:

```
ngx_process_t    ngx_processes[NGX_MAX_PROCESSES];
```

其中,ngx_process_t类型定义为:

```
typedef struct {
    ngx_pid_t           pid;
    int                 status;
    ngx_socket_t        channel[2];

    ngx_spawn_proc_pt   proc;
    void               *data;
    char               *name;

    unsigned            respawn:1;
    unsigned            just_spawn:1;
    unsigned            detached:1;
    unsigned            exiting:1;
    unsigned            exited:1;
} ngx_process_t;
```

在这里,我们只关心成员channel成员，这个两元素的数组即用来存放一个匿名套接字对。

我们假设程序运行后，有1个master进程和4个worker进程。那么，对这5个进程而言，每个进程都有一个4元素的数组ngx_processes[4]，数组中每个元素都是一个ngx_process_t类型的结构体，包含了相应的某个worker进程的相关信息。我们这里关心的是每个结构体的channel数组成员。

绘制成表如下:

ngx_processes数组	master	worker0	worker1	worker2	worker3
ngx_processes[0].channel	[x,x]	[x,x]	[x,x]	[x,x]	[x,x]
ngx_processes[1].channel	[x,x]	[x,x]	[x,x]	[x,x]	[x,x]
ngx_processes[2].channel	[x,x]	[x,x]	[x,x]	[x,x]	[x,x]
ngx_processes[3].channel	[x,x]	[x,x]	[x,x]	[x,x]	[x,x]

上表的每一列表示每个进程的ngx_processes数组的各个元素的channel成员。

其中，master进程列中的每一个元素，表示master进程与对应的每个worker进程之间的匿名套接字对。

而每一个worker进程列中的每一个元素，表示该worker进程与对应的每个worker进程之间的匿名套接字对。当然这只是一个粗略的说法，与真实情况并不完全相符，还有很多细节需要进一步阐述。

我们直接借助《深入剖析Nginx》，直接看下图的实例:

ngx_processes数组	master	worker0	worker1	worker2	worker3
ngx_processes[0].channel	[3,7]	[**-1,7**]	[3,-1]	[3,-1]	[3,-1]
ngx_processes[1].channel	[8,9]	[3,0]	[**-1,9**]	[8,-1]	[8,-1]
ngx_processes[2].channel	[10,11]	[9,0]	[7,0]	[**-1,11**]	[10,-1]
ngx_processes[3].channel	[12,13]	[10,0]	[8,0]	[7,0]	[**-1,13**]

再次感谢《深入剖析Nginx》的作者高群凯，觉得在这里我没法表达的比他更好了。所以下面会引用很多该书中的内容。

在上表中，每一个单元格的内容[a,b]分别表示channel[0]和channel[1]的值，-1表示这之前是描述符，但在之后被主动close()掉了，0表示这一直都无对应的描述符，其他数字表示对应的描述符值。

每一列数据都表示该列所对应进程与其他进程进行通信的描述符，如果当前列所对应进程为父进程，那么它与其它进程进行通信的描述符都为channel[0](其实channel[1]也可以)；如果当前列所对应的进程为子进程，那么它与父进程进行通信的描述符为channel[1]（注：这里书中说的太简略，应该为如果当前列所对应的进程为子进程，那么它与父进程进行通信的描述符

为该进程的ngx_processes数组中，与本进程对应的元素中的channel[1]，在图中即为标粗的对角线部分，即[-1,7],[-1,9],[-1,11],[-1,13]这四对），与其它子进程进行通信的描述符都为本进程的ngx_processes数组中与该其它进程对应元素的channel[0]。

比如，[3,7]单元格表示，如果父进程向worker0发送消息，需要使用channel[0]，即描述符3，实际上channel[1]也可以，它的channel[1]为7，没有被close()关闭掉，但一直也没有被使用，所以没有影响，不过按道理应该关闭才是。

再比如，[-1,7]单元格表示如果worker0向master进程发送消息，需要使用channel[1]，即描述符7，它的channel[0]为-1，表示已经close()关闭掉了（Nginx某些地方调用close()时并没有设置对应变量为-1，这里只是为了更好的说明，将已经close()掉的描述符全部标记为-1）。

越是后生成的worker进程，其ngx_processes数组的元素中，channel[0]与父进程对应的ngx_processes数组的元素中的channel[0]值相同的越多，因为基本都是继承而来，但前面生成的worker进程，其channel[0]是通过进程间调用sendmsg传递获得的，所以与父进程对应的channel[0]不一定相等。比如，如果worker0向worker3发送消息，需要使用worker0进程的ngx_processes[3]元素的channel[0],即描述符10，而对应master进程的ngx_processes[3]元素的channel[0]却是12。虽然它们在各自进程里表现为不同的整型数字，但在内核里表示同一个描述符结构，即不管是worker0往描述符10写数据，还是master往描述符12写数据，worker3都能通过描述符13正确读取到这些数据，至于worker3怎么识别它读到的数据是来自worker0，还是master，就得靠其他收到的数据特征，比如pid，来做标记区分。

关于上段讲的，一个子进程如何区分接收到的数据是来自哪一个进程，我们可以看一下Nginx-1.6.2中的一段代码:

```
ngx_int_t
ngx_write_channel(ngx_socket_t s, ngx_channel_t *ch, size_t size,
    ngx_log_t *log)
{
    ssize_t             n;
    ngx_err_t           err;
    struct iovec        iov[1];
    struct msghdr       msg;

#if (NGX_HAVE_MSGHDR_MSG_CONTROL)

    union {
        struct cmsghdr  cm;
        char            space[CMSG_SPACE(sizeof(int))];
    } cmsg;

    if (ch->fd == -1) {
        msg.msg_control = NULL;
        msg.msg_controllen = 0;

    } else {
        msg.msg_control = (caddr_t) &cmsg;
        msg.msg_controllen = sizeof(cmsg);
```

```
        ngx_memzero(&cmsg, sizeof(cmsg));

        cmsg.cm.cmsg_len = CMSG_LEN(sizeof(int));
        cmsg.cm.cmsg_level = SOL_SOCKET;
        cmsg.cm.cmsg_type = SCM_RIGHTS;

        /*
         * We have to use ngx_memcpy() instead of simple
         *   *(int *) CMSG_DATA(&cmsg.cm) = ch->fd;
         * because some gcc 4.4 with -O2/3/s optimization issues the warning:
         *   dereferencing type-punned pointer will break strict-aliasing rules
         *
         * Fortunately, gcc with -O1 compiles this ngx_memcpy()
         * in the same simple assignment as in the code above
         */

        ngx_memcpy(CMSG_DATA(&cmsg.cm), &ch->fd, sizeof(int));
    }

    msg.msg_flags = 0;

#else

    if (ch->fd == -1) {
        msg.msg_accrights = NULL;
        msg.msg_accrightslen = 0;

    } else {
        msg.msg_accrights = (caddr_t) &ch->fd;
        msg.msg_accrightslen = sizeof(int);
    }

#endif

    iov[0].iov_base = (char *) ch;
    iov[0].iov_len = size;

    msg.msg_name = NULL;
    msg.msg_namelen = 0;
    msg.msg_iov = iov;
    msg.msg_iovlen = 1;

    n = sendmsg(s, &msg, 0);

    if (n == -1) {
        err = ngx_errno;
        if (err == NGX_EAGAIN) {
            return NGX_AGAIN;
        }

        ngx_log_error(NGX_LOG_ALERT, log, err, "sendmsg() failed");
        return NGX_ERROR;
```

```
    }

    return NGX_OK;
}
```

在调用时，参数ch即为发送的数据部分，其类型定义如下:

```
typedef struct {
     ngx_uint_t  command;
     ngx_pid_t   pid;
     ngx_int_t   slot;
     ngx_fd_t    fd;
} ngx_channel_t;
```

可见，其中就包含了发送方的pid。

最后，就目前Nginx代码来看，子进程并没有往父进程发送任何消息，子进程之间也没有相互通信的逻辑。也许是因为Nginx有其他一些更好的进程通信方式，比如共享内存等，所以这种channel通信目前仅作为父进程往子进程发送消息使用。但由于有这个架构在，可以很轻松使用channel机制来完成各进程间的通信任务。

1.3 Nginx中的相关代码流程

下面，将上面所讲的内容，在Nginx代码中的流程，大概梳理一遍。本文所有代码片段，均来自于nginx-1.6.2。

首先是main函数调用ngx_master_process_cycle:

```
    if (ngx_process == NGX_PROCESS_SINGLE) {
        ngx_single_process_cycle(cycle);

    } else {
        ngx_master_process_cycle(cycle);
    }

    return 0;
```

ngx_master_process_cycle调用ngx_start_worker_processes:

```
ccf = (ngx_core_conf_t *) ngx_get_conf(cycle->conf_ctx, ngx_core_module);

ngx_start_worker_processes(cycle, ccf->worker_processes,
                           NGX_PROCESS_RESPAWN);
ngx_start_cache_manager_processes(cycle, 0);
```

在ngx_start_worker_processes函数中，完成对所有worker进程的fork操作:

```
static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
{
    ngx_int_t      i;
    ngx_channel_t  ch;

    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start worker processes");

    ngx_memzero(&ch, sizeof(ngx_channel_t));

    ch.command = NGX_CMD_OPEN_CHANNEL;

    for (i = 0; i < n; i++) {

        ngx_spawn_process(cycle, ngx_worker_process_cycle,
                          (void *) (intptr_t) i, "worker process", type);

        ch.pid = ngx_processes[ngx_process_slot].pid;
        ch.slot = ngx_process_slot;
        ch.fd = ngx_processes[ngx_process_slot].channel[0];

        ngx_pass_open_channel(cycle, &ch);
    }
}
```

上述代码调用的ngx_spawn_process即完成具体的socketpair()操作和fork操作:

```
ngx_pid_t
ngx_spawn_process(ngx_cycle_t *cycle, ngx_spawn_proc_pt proc, void *data,
     char *name, ngx_int_t respawn)
{
       ......
       if (socketpair(AF_UNIX, SOCK_STREAM, 0, ngx_processes[s].channel) == -1)
        {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "socketpair() failed while spawning \"%s\"", name);
            return NGX_INVALID_PID;
        }
        ......
        ngx_channel = ngx_processes[s].channel[1];
        ......
        ngx_process_slot = s;
        pid = fork();
        ......
}
```

在上一段代码中可以看到，master进程在调用socketpair后，将生成的channel[1]保存在全局变量ngx_channel中,ngx_channel全局变量的作用是，子进程中会使用该全局变量，并加入到自己的事件中，达到的效果即是子进程将channel[1]加入到自己的事件中。

话分两头，我们先来具体看看子进程的流程。

在主进程执行完fork之后，ngx_start_worker_processes会调用proc回调:

```
    pid = fork();

     switch (pid) {

     case -1:
         ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                       "fork() failed while spawning \"%s\"", name);
         ngx_close_channel(ngx_processes[s].channel, cycle->log);
         return NGX_INVALID_PID;

     case 0:
         ngx_pid = ngx_getpid();
         proc(cycle, data);
         break;

     default:
         break;
     }
```

其中,proc即为ngx_worker_process_cycle。ngx_worker_process_cycle会调用ngx_worker_process_init函数，子进程将从父进程处继承到的channel[1]加入到自己的事件集中，就是在这个函数中完成的:

```
static void
ngx_worker_process_init(ngx_cycle_t *cycle, ngx_int_t worker)
{
......
   for (n = 0; n < ngx_last_process; n++) {

        if (ngx_processes[n].pid == -1) {
            continue;
        }

        if (n == ngx_process_slot) {
            continue;
        }

        if (ngx_processes[n].channel[1] == -1) {
            continue;
        }

        if (close(ngx_processes[n].channel[1]) == -1) {
            ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                          "close() channel failed");
        }
    }

    if (close(ngx_processes[ngx_process_slot].channel[0]) == -1) {
        ngx_log_error(NGX_LOG_ALERT, cycle->log, ngx_errno,
                      "close() channel failed");
    }

#if 0
    ngx_last_process = 0;
#endif

    if (ngx_add_channel_event(cycle, ngx_channel, NGX_READ_EVENT,
                              ngx_channel_handler)
        == NGX_ERROR)
    {
        /* fatal */
        exit(2);
    }
......
}
```

具体的将channel[1]添加到事件集中的操作，是由ngx_add_channel_event来完成的，对应的回调处理函数为ngx_channel_handler,同时我们看到，在添加之前，还进行了很多close的工作，这就于之前的示例表里，那些描述符为-1的表项相对应了。

此时，子进程已经将从父进程那里继承来的channel[1]加入到了自己的监听事件集中，这样，一个子进程从自己的ngx_processses数组中，对应自己的那一个元素中的channel[1]中，即可读取来自其他进程的消息。收到消息时，将执行设置好的回调函数ngx_channel_handler，把

接收到的新子进程的相关信息存储在自己的全局变量ngx_processes数组内。见下面的代码:

```
static void
ngx_channel_handler(ngx_event_t *ev)
{
    ......
          case NGX_CMD_OPEN_CHANNEL:

            ngx_log_debug3(NGX_LOG_DEBUG_CORE, ev->log, 0,
                           "get channel s:%i pid:%P fd:%d",
                           ch.slot, ch.pid, ch.fd);

            ngx_processes[ch.slot].pid = ch.pid;
            ngx_processes[ch.slot].channel[0] = ch.fd;
            break;
    ......
}
```

我们再回到父进程中。

父进程在从ngx_spawn_process返回后，回来继续执行ngx_start_worker_processes:

```
static void
ngx_start_worker_processes(ngx_cycle_t *cycle, ngx_int_t n, ngx_int_t type)
{
    ngx_int_t      i;
    ngx_channel_t  ch;

    ngx_log_error(NGX_LOG_NOTICE, cycle->log, 0, "start worker processes");

    ngx_memzero(&ch, sizeof(ngx_channel_t));

    ch.command = NGX_CMD_OPEN_CHANNEL;

    for (i = 0; i < n; i++) {

        ngx_spawn_process(cycle, ngx_worker_process_cycle,
                          (void *) (intptr_t) i, "worker process", type);

        ch.pid = ngx_processes[ngx_process_slot].pid;
        ch.slot = ngx_process_slot;
        ch.fd = ngx_processes[ngx_process_slot].channel[0];

        ngx_pass_open_channel(cycle, &ch);
    }
}
```

其中的for循环即表示，父进程会把刚刚生成的子进程的channel[0],放在一条消息的内容中发送给之前生成的子进程。消息的格式定义为:

```
typedef struct {
     ngx_uint_t  command;
     ngx_pid_t   pid;
     ngx_int_t   slot;
     ngx_fd_t    fd;
} ngx_channel_t;
```

我们看下ngx_pass_open_channel函数:

```
static void
ngx_pass_open_channel(ngx_cycle_t *cycle, ngx_channel_t *ch)
{
    ngx_int_t  i;

    for (i = 0; i < ngx_last_process; i++) {

        if (i == ngx_process_slot
            || ngx_processes[i].pid == -1
            || ngx_processes[i].channel[0] == -1)
        {
            continue;
        }

        ngx_log_debug6(NGX_LOG_DEBUG_CORE, cycle->log, 0,
                       "pass channel s:%d pid:%P fd:%d to s:%i pid:%P fd:%d",
                       ch->slot, ch->pid, ch->fd,
                       i, ngx_processes[i].pid,
                       ngx_processes[i].channel[0]);

        /* TODO: NGX_AGAIN */

        ngx_write_channel(ngx_processes[i].channel[0],
                          ch, sizeof(ngx_channel_t), cycle->log);
    }
}
```

从该函数定义中，可以很清晰的看到“往之前生成的每个进程发送消息”。对之前的每个子进程，具体消息发送工作，是由函数ngx_write_channel完成的。

ngx_write_channel函数的第一个参数是之前某个进程从master进程继承来的channel[0],第二个参数发送的内容。其中包含了当前进程的pid,slot号，command等信息，最重要的是，包含了当前子进程的channel[0]，其实是实现了一个简单的协议。注意，当前子进程的channel[0]虽然存在ngx_channel_t类型的消息体中，但真正文件描述符的传递操作，是ngx_write_channel通过发送控制信息来完成的。接收进程虽然在接收到的消息体中获得了发送进程的channel[0]这个值，但并不能直接使用，必须根据控制信息来获取一个新的文件描述符。参看进程间传递文件描述符。

至此，父子进程间的配合，使得所有的子进程均拥有了其他子进程的channel[0]，而另一方面，由于所有子进程的channel[1]已加入到自己的监听事件集，所以子进程之间的通信通道即被建立起来。

值得一提的是，父进程在调用socketpair()产生一个匿名套接字对后，再fork出一个子进程，那么现在有4个文件描述符了。其实对这4个文件描述符中的任何一个进行写入，从其他3个描述符中的任何一个均可以进行读取操作。

但Nginx通过一些close()操作,有意达到这样一种目的:

- 对任何一个子进程，其ngx_processes数组中，对应其它进程的元素,其channel[0]用来向该"其他进程"发送消息。
- 对任何一个子进程，其ngx_processes数组中，对应本进程的元素,其channel[1]用来接收来自其他进程的消息，这个其他进程既包括其他子进程，也包括master进程。至于如何区分是来自哪个进程，以及该消息是用来做什么的，则通过判断ngx_channel_t类型的消息的command,pid,slot等成员来协商。
- 对master进程, 其ngx_processes数组的中，对应相应子进程的元素的channel[0],用来向该子进程发送消息。注:其实channel[1]也可以，但按常理，master进程的ngx_processes数组所有元素的channel[1]应该关闭的。

2. Nginx中的共享内存

2.1 概述

> 共享内存是Linux下提供的最基本的进程间通信方法，它通过mmap或者shmget系统调用在内存中创建了一块连续的线性地址空间，而通过munmap或者shmdt系统调用可以释放这块内存。使用共享内存的好处是当多个进程使用同一块共享内存时，在任何一个进程修改了共享内存中的内容后，其他进程通过访问这段共享内存都能够得到修改后的内容。
>
> 陶辉《深入理解Nginx》

共享内存可以说是最有用的进程间通信方式，也是最快的IPC形式。两个不同进程A、B共享内存的意思是，同一块物理内存被映射到进程A、B各自的进程地址空间。进程A可以即时看到进程B对共享内存中数据的更新，反之亦然。由于多个进程共享同一块内存区域，必然需要某种同步机制，互斥锁和信号量都可以。

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据：一次从输入文件到共享内存区，另一次从共享内存区到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，

有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

Linux中，共享内存可以通过两个系统系统调用来获得，mmap和shmget，分别属于不同的标准，这不在本文的关注范围之内。mmap语义上比shmget更通用，因为它最一般的做法，是将一个打开的实体文件，映射到一段连续的内存中，各个进程可以根据各自的权限对该段内存进行相应的读写操作，其他进程则可以看到其他进程写入的结果。而shmget在语义上相当于是匿名的mmap，即不关注实体文件，直接在内存中开辟这块共享区域，mmap通过设置调用时的参数，也可达到这种效果，一种方法是映射/dev/zero设备,另一种是使用MAP_ANON选项。至于mmap和shmget的效率，跟不同的内核实现相关，不在本文关注范围内。

除了上面的简单描述外，本文不打算仔细介绍mmap和shmget的使用。有如下相关资料可以参考:

1. Linux环境进程间通信（五）: 共享内存（上）
2. Linux环境进程间通信（五）: 共享内存（下）
3. APUE,14.8,15.9

2.2 Nginx中的实现

那么，在Nginx中，到底是选用mmap映射到/dev/null，还是使用MAP_ANON选项调用mmap，或者是使用shmget呢？看相关实现的代码就会一目了然:

```
/*
 * Copyright (C) Igor Sysoev
 * Copyright (C) Nginx, Inc.
 */


#include <ngx_config.h>
#include <ngx_core.h>


#if (NGX_HAVE_MAP_ANON)

ngx_int_t
ngx_shm_alloc(ngx_shm_t *shm)
{
    shm->addr = (u_char *) mmap(NULL, shm->size,
                                PROT_READ|PROT_WRITE,
                                MAP_ANON|MAP_SHARED, -1, 0);

    if (shm->addr == MAP_FAILED) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "mmap(MAP_ANON|MAP_SHARED, %uz) failed", shm->size);
        return NGX_ERROR;
```

```
    }

    return NGX_OK;
}


void
ngx_shm_free(ngx_shm_t *shm)
{
    if (munmap((void *) shm->addr, shm->size) == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "munmap(%p, %uz) failed", shm->addr, shm->size);
    }
}

#elif (NGX_HAVE_MAP_DEVZERO)

ngx_int_t
ngx_shm_alloc(ngx_shm_t *shm)
{
    ngx_fd_t  fd;

    fd = open("/dev/zero", O_RDWR);

    if (fd == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "open(\"/dev/zero\") failed");
        return NGX_ERROR;
    }

    shm->addr = (u_char *) mmap(NULL, shm->size, PROT_READ|PROT_WRITE,
                                MAP_SHARED, fd, 0);

    if (shm->addr == MAP_FAILED) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "mmap(/dev/zero, MAP_SHARED, %uz) failed", shm->size);
    }

    if (close(fd) == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "close(\"/dev/zero\") failed");
    }

    return (shm->addr == MAP_FAILED) ? NGX_ERROR : NGX_OK;
}


void
ngx_shm_free(ngx_shm_t *shm)
{
    if (munmap((void *) shm->addr, shm->size) == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "munmap(%p, %uz) failed", shm->addr, shm->size);
```

```
    }
}

#elif (NGX_HAVE_SYSVSHM)

#include <sys/ipc.h>
#include <sys/shm.h>


ngx_int_t
ngx_shm_alloc(ngx_shm_t *shm)
{
    int  id;

    id = shmget(IPC_PRIVATE, shm->size, (SHM_R|SHM_W|IPC_CREAT));

    if (id == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "shmget(%uz) failed", shm->size);
        return NGX_ERROR;
    }

    ngx_log_debug1(NGX_LOG_DEBUG_CORE, shm->log, 0, "shmget id: %d", id);

    shm->addr = shmat(id, NULL, 0);

    if (shm->addr == (void *) -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno, "shmat() failed");
    }

    if (shmctl(id, IPC_RMID, NULL) == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "shmctl(IPC_RMID) failed");
    }

    return (shm->addr == (void *) -1) ? NGX_ERROR : NGX_OK;
}


void
ngx_shm_free(ngx_shm_t *shm)
{
    if (shmdt(shm->addr) == -1) {
        ngx_log_error(NGX_LOG_ALERT, shm->log, ngx_errno,
                      "shmdt(%p) failed", shm->addr);
    }
}

#endif
```

上面的代码即是Nginx源代码中的src/os/unix/ngx_shemem.c的全部内容。可见，整个文件只是为了提供两个接口：ngx_shm_alloc和ngx_shm_free。而这两个接口的实现，按如下逻辑来决定:

1. 如果当前系统的mmap系统调用支持MAP_ANON选项，则使用带MAP_ANON选项的mmap。
2. 如果1不满足，则如果当前系统mmap系统调用支持映射/dev/zero设备，则使用mmap映射/dev/zero的方式来实现。
3. 如果上面1和2都不满足，且如果当前系统支持shmget系统调用的话，则使用该系统调用来实现。

看到这里，也许大家就有疑问了，如果当前3个条件都不满足怎么办，那就没辙了，ngx_shm_alloc接口没有相应的定义，只能在链接的时候就不成功了。

另外，关于上面三种情况的判断，都是通过相应的宏是否定义来进行的，而相应的宏的定义，是在auto/unix脚本中进行的，该脚本会写一端测试程序来判断相应的系统调用是否支持，如果支持，则在configure后自动生成的objs/ngx_auto_config.h文件中定义对应的宏。

3. channel机制和共享内存在Nginx中的使用情况

前面讲Nginx中的channel机制时提到，Nginx虽然提供了这种机制，但目前很少用到，而共享内存却相对用的比较多了。例如，为了统计Nginx总体的http请求处理情况，需要跨越多个worker来计算，Nginx自带的http模块ngx_http_stub_status_module即主要依赖共享内存的方式。

Nginx如何控制某个特性是否打开

提到Nginx，大家首先会想到它的高性能，异步框架、模块化、upstream、红黑树等耳熟能详的技术实现。这些确实也是Nginx的核心，但作为一个优秀的开源项目，Nginx可以供我们借鉴的远不止这些，例如本文的话题：如何控制某个特性是否打开？

我们知道，在Linux下用源码安装方式编译安装一个软件时，标准情况下是有一个configure的动作，这个动作即是在编译前对环境进行检查，服务于后面的编译和安装，Nginx当然也不例外。

Nginx的configure文件是一个入口，在里面调用了很多其他脚本，这些脚本都位于源代码的auto目录下。本文重点涉及其中两个脚本：auto/have和auto/define.

它们的内容极其简单，分别如下： 1.auto/have:

```
# Copyright (C) Igor Sysoev
# Copyright (C) Nginx, Inc.


cat << END >> $NGX_AUTO_CONFIG_H

#ifndef $have
#define $have 1
#endif

END
```

2.auto/define

```
# Copyright (C) Igor Sysoev
# Copyright (C) Nginx, Inc.


cat << END >> $NGX_AUTO_CONFIG_H

#ifndef $have
#define $have $value
#endif

END
```

可见，两个脚本只有一处不同，是将have变量定义成value变量还是将其定义为1，所以后面仅仅以have脚本为例进行说明。

have脚本的用法：

```
have=XXXX  . auto/have
```

就会在$NGX_AUTO_CONFIG_H所代表的文件里（默认为objs/ngx_auto_config.h)追加如下内容：

```
#ifndef XXXX
#define XXXX 1
#endif
```

这样，就可以用XXXX宏是否定义来控制编译的过程。

下面以ngx_memalign这个Nginx内部接口为例来进行详细的说明。 ngx_memalign是Nginx里最基本的接口之一，经常会被调用。从接口的名字就可以看出，这个接口是用来处理内存分配和对齐的。其定义在ngx_alloc.h中：

```
#if (NGX_HAVE_POSIX_MEMALIGN || NGX_HAVE_MEMALIGN)
void *ngx_memalign(size_t alignment, size_t size, ngx_log_t *log);
#else
#define ngx_memalign(alignment, size, log) ngx_alloc(size, log)
#endif
```

上面代码的意图是，如果定义了NGX_HAVE_POSIX_MEMALIGN宏或者NGX_HAVE_MEMALIGN宏，则声明函数ngx_memalign，否则，简单的对ngx_alloc进行一下封装（ngx_alloc是对malloc的简单封装）。

我们在来看函数ngx_memalign在ngx_alloc.c中的定义：

```
#if (NGX_HAVE_POSIX_MEMALIGN)

void *
ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
{
    void *p;
    int err;

    err = posix_memalign(&p, alignment, size);

    if (err) {
        ngx_log_error(NGX_LOG_EMERG, log, err,
                      "posix_memalign(%uz, %uz) failed", alignment, size);
        p = NULL;
    }

    ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
                   "posix_memalign: %p:%uz @%uz", p, size, alignment);

    return p;
}

#elif (NGX_HAVE_MEMALIGN)

void *
ngx_memalign(size_t alignment, size_t size, ngx_log_t *log)
{
    void *p;

    p = memalign(alignment, size);
    if (p == NULL) {
        ngx_log_error(NGX_LOG_EMERG, log, ngx_errno,
                      "memalign(%uz, %uz) failed", alignment, size);
    }

    ngx_log_debug3(NGX_LOG_DEBUG_ALLOC, log, 0,
                   "memalign: %p:%uz @%uz", p, size, alignment);

    return p;
}

#endif
```

上面代码的意图为，如果定义了NGX_HAVE_POSIX_MEMALIGN，那么ngx_memalign就是对posix_memalign的简单封装，否则，如果定义了NGX_HAVE_MEMALIGN，则ngx_memalign是对memalign的简单封装。

那么NGX_HAVE_POSIX_MEMALIGN和NGX_HAVE_MEMALIGN又是什么时候被定义呢？

过程如下： 在auto/unix脚本，列出需要检查的接口，这里就是memalign和posix_memalign了，交给auto/feature脚本来逐一处理，auto/feature脚本会对针对每一个带检查的接口，生成一个最基本的临时c源文件，该源文件会调用该接口。feature脚本然后对该源文件进行编译并判断最终生成的目标文件的可执行性。用这种动态检测的方法来判断该接口在当前系统中是否支持。

如果feature脚本验证出posix_memalign和memalign接口在当前系统中都可用后，则会逐一调用have脚本：

```
have=$ngx_have_feature . auto/have
```

这里，变量ngx_have_feature即是NGX_HAVE_POSIX_MEMALIGN和NGX_HAVE_MEMALIGN。

最终，have在configure、auto/unix、auto/feature、auto/have各个脚本的通力合作下，在objs/ngx_auto_config.h中就有了对NGX_HAVE_POSIX_MEMALIGN和NGX_HAVE_MEMALIGN的定义，进而影响到ngx_memalign接口的实现。

让我们从需求出发，将接口ngx_memalign的需求描述一遍：

1. 如果系统支持posix_memalign，则ngx_memalign是对posix_memalign的简单封装。
2. 如果系统不支持posix_memalign，但支持memalign，则ngx_memalign是对memalign的简单封装。
3. 如果系统竟然对posix_memalign和memalign都不支持，则ngx_memalign是对malloc的简单封装。

实现需求很简单，但如何实现的优雅，则是另一回事。Nginx向我们展示了，一个在C上尽力把服务器性能做到极致的项目，是如何在脚本上也尽最大努力做得漂亮。

用proxy_intercept_errors和recursive_error_pages代理多次302

302是HTTP协议中的一个经常被使用状态码，是多种重定向方式的一种，其语义经常被解释为“Moved Temporarily”。这里顺带提一下，现实中用到的302多为误用（与303，307混用），在HTTP/1.1中，它的语义为“Found”.

302有时候很明显，有时候又比较隐蔽。最简单的情况，是当我们在浏览器中输入一个网址A，然后浏览器地址栏会自动跳到B，进而打开一个网页，这种情况就很可能是302。

比较隐蔽的情况经常发生在嵌入到网页的播放器中。例如，当你打开一个优酷视频播放页面时，抓包观察一下就会经常发现302的影子。但由于这些url并不是直接在浏览器中打开的，所以在浏览器的地址栏看不到变化，当然，如果将这些具体的url特意挑出来复制到浏览器地址栏里，还是可以观察到的。

上一段提到了优酷。其实现在多数在线视频网站都会用到302，原因很简单，视频网站流量一般较大，都会用到CDN,区别只在于是用自建CDN还是商业CDN。而由于302的重定向语义（再重复一遍，302的语义广泛的被误用，在使用302的时候，我们很可能应该使用303或307，但后面都不再纠结这一点），可以与CDN中的调度很好的结合起来。

我们来看一个例子，打开一个网易视频播放页面，抓一下包，找到302状态的那个url。例如：

```
http://flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J05P-mobile.mp4
```

我们把它复制到浏览器地址栏中，会发现地址栏迅速的变为了另外一个url，这个Url是不定的，有可能为：

```
http://14.18.140.83/f6c00af500000000-1408987545-236096587/data6/flv.bn.netease.com/tvmrep
```

用curl工具会更清楚的看到整个过程：

```
curl -I "http://flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J05P-mobile.mp4"
HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Mon, 25 Aug 2014 14:49:43 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
NG: CCN-SW-1-5L2
X-Mod-Name: GSLB/3.1.0
Location: http://119.134.254.9/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J0

HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Mon, 25 Aug 2014 14:49:41 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
X-Mod-Name: Mvod-Server/4.3.3
Location: http://119.134.254.7/cc89fdac00000000-1408983581-2095617481/data4/flv.bn.neteas
NG: CHN-SW-1-3Y1

HTTP/1.1 200 OK
Server: nginx
Date: Mon, 25 Aug 2014 14:49:41 GMT
Content-Type: video/mp4
Content-Length: 3706468
Last-Modified: Mon, 25 Aug 2014 00:23:50 GMT
Connection: keep-alive
Cache-Control: no-cache
ETag: "53fa8216-388e64"
NG: CHN-SW-1-3g6
X-Mod-Name: Mvod-Server/4.3.3
Accept-Ranges: bytes
```

可以看到，这中间经历了两次302。

先暂时将这个例子放在一边，再来说说另一个重要的术语：proxy.我们通常会戏称，某些领导是302类型的，某些领导是proxy类型的。302类型的领导，一件事情经过他的手，会迅速的转给他人，而proxy类型的领导则会参与到事情中来，甚至把事情全部做完。

回到上面的例子，如果访问一个url中途会有多个302，那如果需要用Nginx设计一个proxy，来隐藏掉中间所有的这些302，该怎么做呢

1.原始Proxy

我们知道，Nginx本身就是一个优秀的代理服务器。因此，首先我们来架设一个Nginx正向代理，服务器IP为192.168.109.128（我的一个测试虚拟机）。

初始配置简化如下：

```
server {
        listen 80;
        location / {
                rewrite_by_lua '
                        ngx.exec("/proxy-to" .. ngx.var.request_uri)
                ';
        }

        location ~ /proxy-to/([^/]+)(.*) {
                proxy_pass http://$1$2$is_args$query_string;

        }
}
```

实现的功能是，当使用

```
http://192.168.109.128/xxxxxx
```

访问该代理时，会proxy到xxxxxx所代表的真实服务器。

测试结果如下：

```
curl -I "http://192.168.109.128/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.4.6
Date: Mon, 25 Aug 2014 14:50:54 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
NG: CCN-SW-1-5L2
X-Mod-Name: GSLB/3.1.0
Location: http://183.61.140.24/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J0

HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Mon, 25 Aug 2014 14:50:55 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
X-Mod-Name: Mvod-Server/4.3.3
Location: http://183.61.140.20/540966e500000000-1408983655-236096587/data1/flv.bn.netease
NG: CHN-ZJ-4-3M4

HTTP/1.1 200 OK
Server: nginx
Date: Mon, 25 Aug 2014 14:50:55 GMT
Content-Type: video/mp4
Content-Length: 3706468
Last-Modified: Mon, 25 Aug 2014 00:31:03 GMT
Connection: keep-alive
Cache-Control: no-cache
ETag: "53fa83c7-388e64"
NG: CHN-ZJ-4-3M4
X-Mod-Name: Mvod-Server/4.3.3
Accept-Ranges: bytes
```

可见，虽然使用proxy，但过程与原始访问没有什么区别。访问过程为，当访问

```
http://192.168.109.128/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J05P-mobil
```

时，Nginx会将该请求proxy到

```
http://flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J05P-mobile.mp4
```

而后者马上就会返回一个302，所以Nginx作为proxy，将该302传回到客户端，客户端重新发起请求，进而重复之前的多次302.这里说明一个问题，一旦Nginx的proxy的后端返回302后，客户端即与Nginx这个proxy脱离关系了，Nginx无法起到完整的代理的作用。

2. 第1次修改

将配置文件修改为：

```
server {
        listen 80;
        location / {
                rewrite_by_lua '
                        ngx.exec("/proxy-to" .. ngx.var.request_uri)
                ';
        }

        location ~ /proxy-to/([^/]+)(.*) {
                proxy_pass http://$1$2$is_args$query_string;
                error_page 302 = @error_page_302;

        }
        location @error_page_302 {
                rewrite_by_lua '
                        local _, _, upstream_http_location = string.find(ngx.var.upstream
                        ngx.header["zzzz"] = "/proxy-to" .. upstream_http_location
                        ngx.exec("/proxy-to" .. upstream_http_location);
                ';

        }
}
```

与上面的区别在于，使用了一个error_page，目的是当发现proxy的后端返回302时，则用这个302的目的location继续proxy，而不是直接返回给客户端。并且这个逻辑里面包含着递归的意思，一路跟踪302，直到最终返回200的那个地址。测试结果如下：

```
curl -I "http://192.168.109.128/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.4.6
Date: Mon, 25 Aug 2014 15:01:17 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
NG: CCN-SW-1-5L2
X-Mod-Name: GSLB/3.1.0
Location: http://183.61.140.24/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J0

HTTP/1.1 302 Moved Temporarily
Server: nginx
Date: Mon, 25 Aug 2014 15:01:17 GMT
Content-Type: text/html
Content-Length: 154
Connection: keep-alive
X-Mod-Name: Mvod-Server/4.3.3
Location: http://183.61.140.20/a90a952900000000-1408984277-236096587/data1/flv.bn.netease
NG: CHN-ZJ-4-3M4

HTTP/1.1 200 OK
Server: nginx
Date: Mon, 25 Aug 2014 15:01:17 GMT
Content-Type: video/mp4
Content-Length: 3706468
Last-Modified: Mon, 25 Aug 2014 00:31:03 GMT
Connection: keep-alive
Cache-Control: no-cache
ETag: "53fa83c7-388e64"
NG: CHN-ZJ-4-3M4
X-Mod-Name: Mvod-Server/4.3.3
Accept-Ranges: bytes
```

可见，本次修改仍然没有成功！为什么呢？分析一下，我们在@error_page_302这个location里已经加了一个头部打印语句，可是在测试中，该头部并没有打出来，可见流程并没有进入到@error_page_302这个location。

原因在于

```
error_page 302 = @error_page_302;
```

error_page默认是本次处理的返回码。作为proxy，本次处理，只要转发上游服务器的响应成功，应该状态码都是200.即，我们真正需要检查的，是proxy的后端服务器返回的状态码，而不是proxy本身返回的状态码。查一下Nginx的wiki,proxy_intercept_errors指令正是干这个的:

```
Syntax:    proxy_intercept_errors on | off;
Default:
proxy_intercept_errors off;
Context:    http, server, location
Determines whether proxied responses with codes greater than or equal to 300 should be pa
```

3. 第二次修改

```
server {
        listen 80;
        proxy_intercept_errors on;
        location / {
                rewrite_by_lua '
                        ngx.exec("/proxy-to" .. ngx.var.request_uri)
                ';
        }
        location ~ /proxy-to/([^/]+)(.*) {
                proxy_pass http://$1$2$is_args$query_string;
                error_page 302 = @error_page_302;

        }
        location @error_page_302 {
                rewrite_by_lua '
                        local _, _, upstream_http_location = string.find(ngx.var.upstream
                        ngx.header["zzzz"] = "/proxy-to" .. upstream_http_location
                        ngx.exec("/proxy-to" .. upstream_http_location);
                ';
        }
}
```

与上一次修改相比，区别仅仅在于增加了一个proxy_intercept_errors指令。测试结果如下：

```
curl -I "http://192.168.109.128/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J
HTTP/1.1 302 Moved Temporarily
Server: nginx/1.4.6
Date: Mon, 25 Aug 2014 15:05:54 GMT
Content-Type: text/html
Content-Length: 160
Connection: keep-alive
zzzz: /proxy-to/183.61.140.24/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J05
```

这次更神奇了，直接返回一个302状态完事，也不继续跳转了。问题出在，虽然第一次302，请求成功的进入到@error_page_302，但后续的error_page指令却没起作用。也就是说，error_page只检查了第一次后端返回的状态码，而没有继续检查后续的后端状态码。

查一下资料，这个时候，另一个指令 recursive_error_pages就派上用场了。

4. 第3次修改

```
server {
        listen 80;
        proxy_intercept_errors on;
        recursive_error_pages on;
        location / {
                rewrite_by_lua '
                        ngx.exec("/proxy-to" .. ngx.var.request_uri)
                ';
        }
        location ~ /proxy-to/([^/]+)(.*) {
                proxy_pass http://$1$2$is_args$query_string;
                error_page 302 = @error_page_302;

        }
        location @error_page_302 {
                rewrite_by_lua '
                        local _, _, upstream_http_location = string.find(ngx.var.upstream
                        ngx.header["zzzz"] = "/proxy-to" .. upstream_http_location
                        ngx.exec("/proxy-to" .. upstream_http_location);
                ';
        }
}
```

与上一次相比，仅仅增加了recursive_error_pages on这条指令。测试结果如下：

```
curl -I "http://192.168.109.128/flv.bn.netease.com/tvmrepo/2014/8/5/P/EA3I1J05P/SD/EA3I1J
HTTP/1.1 200 OK
Server: nginx/1.4.6
Date: Mon, 25 Aug 2014 15:09:04 GMT
Content-Type: video/mp4
Content-Length: 3706468
Connection: keep-alive
zzzz: /proxy-to/14.18.140.83/f48bad0100000000-1408984745-236096587/data6/flv.bn.netease.c
Last-Modified: Mon, 25 Aug 2014 00:21:07 GMT
Cache-Control: no-cache
ETag: "53fa8173-388e64"
NG: CHN-MM-4-3FE
X-Mod-Name: Mvod-Server/4.3.3
Accept-Ranges: bytes
```

可见，Nginx终于成功的返回200了。此时，Nginx才真正起到了一个Proxy的功能，隐藏了一个请求原本的多个302链路，只返回客户端一个最终结果。

5. 小结

综上，通过proxy_pass、error_page、proxy_intercept_errors、recursive_error_pages这几个指令的配合使用，可以向客户端隐藏一条请求的跳转细节，直接返回用户一个状态码为200的最终结果。

奇怪的是，在Nginx的官方wiki中并没有recursive_error_pages指令的相关说明。

ngx.var.arg与ngx.req.get_uri_args的区别

ngx.var.arg_xx与ngx.req.get_uri_args["xx"]两者都是为了获取请求uri中的参数，例如

```
http://pureage.info?strider=1
```

为了获取输入参数strider，以下两种方法都可以：

1. local strider = ngx.var.arg_strider
2. local strider = ngx.req.get_uri_args["strider"]

差别在于，当请求uri中有多个同名参数时,ngx.var.arg_xx的做法是取第一个出现的值,ngx.req_get_uri_args["xx"]的做法是返回一个table，该table里存放了该参数的所有值，例如,当请求uri为：

```
http://pureage.info?strider=1&strider=2&strider=3&strider=4
```

时，ngx.var.arg_strider的值为"1",而ngx.req.get_uri_args["strider"]的值为table ["1", "2", "3", "4"]。因此，ngx.req.get_uri_args属于ngx.var.arg_的增强。

ngx.var.arg_的实现是直接使用nginx原生的变量支持，nginx相关代码为：

```
ngx_http_variable_value_t *
ngx_http_get_variable(ngx_http_request_t *r, ngx_str_t *name, ngx_uint_t key)
{
....(略) ....
if (ngx_strncmp(name->data, "arg_", 4) == 0) {

        if (ngx_http_variable_argument(r, vv, (uintptr_t) name) == NGX_OK) {
            return vv;
        }

        return NULL;
    }

    vv->not_found = 1;

    return vv;
}

static ngx_int_t
ngx_http_variable_argument(ngx_http_request_t *r, ngx_http_variable_value_t *v,
    uintptr_t data)
{
    ngx_str_t *name = (ngx_str_t *) data;
```

```
    u_char *arg;
    size_t len;
    ngx_str_t value;

    len = name->len - (sizeof("arg_") - 1);
    arg = name->data + sizeof("arg_") - 1;

    if (ngx_http_arg(r, arg, len, &value) != NGX_OK) {
        v->not_found = 1;
        return NGX_OK;
    }

    v->data = value.data;
    v->len = value.len;
    v->valid = 1;
    v->no_cacheable = 0;
    v->not_found = 0;

    return NGX_OK;
}

ngx_int_t
ngx_http_arg(ngx_http_request_t *r, u_char *name, size_t len, ngx_str_t *value)
{
    u_char *p, *last;

    if (r->args.len == 0) {
        return NGX_DECLINED;
    }

    p = r->args.data;
    last = p + r->args.len;

    for ( /* void */ ; p < last; p++) {

        /* we need '=' after name, so drop one char from last */

        p = ngx_strlcasestrn(p, last - 1, name, len - 1);

        if (p == NULL) {
            return NGX_DECLINED;
        }

        if ((p == r->args.data || *(p - 1) == '&') && *(p + len) == '=') {

            value->data = p + len + 1;

            p = ngx_strlchr(p, last, '&');

            if (p == NULL) {
                p = r->args.data + r->args.len;
            }
```

```
            value->len = p - value->data;

            return NGX_OK;
        }
    }

    return NGX_DECLINED;
}
```

关于nginx中的host变量

关于变量host，在Nginx的官网wiki中是如下说明的：

> $host：in this order of precedence: host name from the request line, or host name from the “Host” request header field, or the server name matching a request

直白的翻译一下:host变量的值按照如下优先级获得：

1. 请求行中的host.
2. 请求头中的Host头部.
3. 与一条请求匹配的server name.

很清楚，有三点，取优先级最高的那个。仅从字面意思上来理解，这个选择的过程为：如果请求行中有host信息，则以请求行中的host作为host变量的值（host与host变量不是一个东西，很拗口）；如果请求行中没有host信息，则以请求头中的Host头的值作为host变量的值；如果前面两者都没有，那么host变量就是与该请求匹配所匹配的serve名。

为了表示语气的加强，再重复一遍，这个规则包含了三个步骤。

但由于某种不知名的原因，网上能找到的大部分关于该变量的说明都只包含两点。

例如这个：

> $host
>
> 请求中的Host字段，如果请求中的Host字段不可用，则为服务器处理请求的服务器名称。

再例如这个：

> $host, 请求信息中的"Host"，如果请求中没有Host行，则等于设置的服务器名;

这些说法都是错误的。它们将一个本来很有内涵的东西向读者隐藏起来，如果你只是一扫而过，在心里默念，哦，这个变量原来是这样的，很简单嘛，那就会错过很多东西。下面就来详细的说明一下该变量。

1.什么是请求行中的host?

我们知道，HTTP是一个文本协议，建立在一个可靠的传输层协议之上。这个传输层协议要是可靠的，面向连接的。由于TCP的普及程度，让它成了HTTP下层协议事实上的标准。但我们要知道，HTTP并不仅限于建立在TCP之上。只要是可靠的，面向连接的传输层协议，都可以用来传输HTTP。下面所说的HTTP，都是指搭载在TCP之上的HTTP。

一个HTTP请求过程是这样的，客户端先与服务器建立起TCP连接，然后再与服务器端进行请求和回复的收发。请求包含请求行、请求头和请求体，其中，根据请求方法的不同，请求体是可选的。

下面开始说请求行。

在发送请求行之前，客户端与服务器已经建立了连接。所以此时请求行中并不需要有服务器的信息。例如，可以如下：

> GET /index.php HTTP/1.1

这就是一个完整的HTTP请求行。虽然请求行中不需要有服务器的信息，但仍然可以在请求行中包含服务器的信息。例如：

> GET www.pureage.info/index.php HTTP/1.1

两者一比较，就很容易理解什么叫请求行中的host了。第一个请求行中，就没有host，第二种请求行中，就带了host，为wwww.pureage.info。

2.Host请求头与HTTP/1.0、HTTP/1.1

一个请求，请求行下面就是一些列的请求头。这些请求头，在HTTP/1.0中，都是可选的，且HTTP/1.0不支持Host请求头；而在**HTTP/1.1**中，**Host**请求头部必须存在。

除了阅读RFC2616来确认这一点外，我们来看看Nginx中的代码：

```
ngx_int_t
ngx_http_process_request_header(ngx_http_request_t *r)
...(略)...
if (r->headers_in.host == NULL && r->http_version > NGX_HTTP_VERSION_10) {
        ngx_log_error(NGX_LOG_INFO, r->connection->log, 0,
                      "client sent HTTP/1.1 request without \"Host\" header");
        ngx_http_finalize_request(r, NGX_HTTP_BAD_REQUEST);
        return NGX_ERROR;
...(略)...
```

即，如果协议版本在HTTP/1.0之上，且请求头部没有Host的话，会直接返回一个Bad Request错误响应。

3.什么是与请求匹配的server name?

server name是指在Nginx配置文件中，在server块中，用server_name指令设置的值。一个server可以多次使用server_name指令，来实现俗称的“虚拟主机”。例如：

```
server {
    listen      80;
    server_name example.org www.example.org;
    ...
}

server {
    listen      80;
    server_name example.net www.example.net;
    ...
}

server {
    listen      80;
    server_name example.com www.example.com;
    ...
}
```

关于虚拟主机的确定方法，还是引用Nginx的官方文档：

> 在这个配置中，nginx仅仅检查请求的“Host”头以决定该请求应由哪个虚拟主机来处理。如果Host头没有匹配任意一个虚拟主机，或者请求中根本没有包含Host头，那nginx会将请求分发到定义在此端口上的默认虚拟主机。在以上配置中，第一个被列出的虚拟主机即nginx的默认虚拟主机——这是nginx的默认行为。而且，可以显式地设置某个主机为默认虚拟主机，即在"listen"指令中设置"default_server"参数：
>
> server { listen 80 default_server; server_name example.net www.example.net; ... }

4.整理一下

上面所有的说明，都仅仅是解释了Nginx官方文档中关于host变量选择的三个来源值。不过，知道了这三个值是什么，怎么来的，以及HTTP/1.0与HTTP/1.1的区别，我们就能彻底搞清楚这个host变量是怎么确定的了。

首先，无论是HTTP/1.0还是HTTP/1.1请求，只要请求行中带了主机信息，那么host变量就是该请求行中所带的host。需要注意的是，host变量是不带端口号的。

所以，下面两条请求的host变量都是www.pureage.info:

> GET http://www.pureage.info/index.php HTTP/1.0
>
> GET http://www.pureage.info/index.php HTTP/1.1

其次，假设请求行中不带主机信息，那么我们就来看请求头部中的Host头。此时HTTP/1.0请求和HTTP/1.1请求的表现就大不相同了。有如下几种情况：

1.HTTP/1.0请求，不带Host头

> GET /index.php HTTP/1.0

此时，host变量为与该请求匹配的虚拟主机的主机名，及在nginx配置文件中与之匹配的server段中server_name指令设置的值。如果该server中没有使用 `server_name` 指令，那么host变量就是一个空值，不存在。

2.HTTP/1.0请求，带Host头

> GET /index.php HTTP/1.0 Host: www.pureage.info

此时，host变量即为www.pureage.info

3.HTTP/1.1请求，不带Host头

> GET /index.php HTTP/1.1

如前所述，HTTP/1.1请求必须携带Host头部。所以这种情况，请求会返回一个Bad Request错误响应。host变量当然就更不存在了。

4.HTTP/1.1请求，带Host头

> GET /index.php HTTP/1.1 Host: www.pureage.info

此时，host变量即为www.pureage.info

通过上面的例子，可以看出，对于HTTP/1.1请求，host变量不会为空，它要么在请求行中指定，要么在Host头部指定，要么就是一条错误的请求；而对于HTTP/1.0请求，host变量有为空的情况，即请求行和请求头中没有指定host，且匹配的server也没有名称时。

5.验证

可以搭建一台Nginx服务器，用telnet来验证上面的说法。此处略过，但这个操作很有必要。

关于proxy_pass的参数路径问题

由于工作需要，开始分析nginx的proxy模块，在分析之前，当然要先会用了。于是开始熟悉该模块的一些指令，其中最基本的指令要属proxy_pass了。nginx的英文文档总是看着感觉有些别扭，于是按惯例先google了一些文章。

这一搜，就掉进坑里了。

这些文章里都把proxy_pass的目标地址是形如“127.0.0.1:8090”和“127.0.0.1:8090/”分开讨论，认为后者“/"的作用是删除url中匹配的部分，然后再讨论目标地址中带了uri的情况。

其实根本没这么复杂，只有两种情况：

（1）目标地址中不带uri。即proxy_pass的参数形如"http://127.0.0.1:8090"。 此时新的目标url中，匹配的uri部分不做修改，原来是什么样就是什么样。

（2）目标地址中带uri。即proxy_pass的参数形如“http://127.0.0.1:8090/dir1/dir2"

此时新的目标url中，匹配的uri部分将会被修改为该参数中的uri，
如"http://127.0.0.1:8888/dir1/dir2."

有人说，你没有讨论ip和端口后带不带”/“的区别。其实是不需要的，因为"/"本身就是一种uri，很明显属于上面的第二种情况，只不过是把原来的uri修改为了现在的uri（”/"），看上去，像是删除了原url中匹配的部分。如果不理解这一点，就会总想着去牢记、区分结尾带不带"/"的情况。 官方文档也是这么叙述的，根本没有提及半句“/"：

> A request URI is passed to the server as follows:
>
> If the proxy_pass directive is specified with a URI, then when a request is passed to the server, the part of a normalized request URI matching the location is replaced by a URI specified in the directive: location /name/ { proxy_pass http://127.0.0.1/remote/; } If proxy_pass is specified without a URI, the request URI is passed to the server in the same form as sent by a client when the original request is processed, or the full normalized request URI is passed when processing the changed URI: location /some/path/ { proxy_pass http://127.0.0.1; }

测试部分如下。

如果配置为：

```
server {
                listen 9090;
                access\_log /home/strider/project/nginx/nginx-1.4.2/log/access\_9090.log;
                location /test1/test2/{
                    proxy\_pass http://127.0.0.1:8090;
                }
        }
```

则有如下对应关系：

```
127.0.0.1:9090/test1/test2/echo1----->127.0.0.1:8090/test1/test2/echo1
127.0.0.1:9090/test1/test2/---->127.0.0.1:8090/test1/test2
```

如果配置为：

```
server {
                listen 9090;
                access\_log /home/strider/project/nginx/nginx-1.4.2/log/access\_9090.log;
                location /test1/test2/{
                    proxy\_pass http://127.0.0.1:8090/;
                }
}
```

则有如下对应关系：

```
127.0.0.1:9090/test1/test2/echo1----->127.0.0.1:8090/echo1
127.0.0.1:9090/test1/test2/---->127.0.0.1:8090/
```

如果配置为：

```
server {
                listen 9090;
                access\_log /home/strider/project/nginx/nginx-1.4.2/log/access\_9090.log;
                location /test1/test2/{
                        proxy\_pass http://127.0.0.1:8090/test1;
                }
}
```

则有如下对应关系：

```
127.0.0.1:9090/test1/test2/echo1----->127.0.0.1:8090/test1echo1
127.0.0.1:9090/test1/test2/---->127.0.0.1:8090/test1
```

如果配置为：

```
server {
                listen 9090;
                access\_log /home/strider/project/nginx/nginx-1.4.2/log/access\_9090.log;
                location /test1/test2/{
                        proxy\_pass http://127.0.0.1:8090/test3/test4/test5;
                }
}
```

则有如下对应关系：

```
127.0.0.1:9090/test1/test2/echo1----->127.0.0.1:8090/test3/test4/test5echo1
127.0.0.1:9090/test1/test2/---->127.0.0.1:80990/test3/test4/test5
```

Nginx的配置解析系统

几个重要的指令类型属性

我们知道，每一个Nginx的模块就是对应一个ngx_module_t类型的结构体，该结构体的commands成员是一个数组，里面包括了该模块的配置指令。每一个commands成员是ngx_command_t类型的结构体，其定义如下:

```
struct ngx_command_s {
    ngx_str_t             name;
    ngx_uint_t            type;
    char               *(*set)(ngx_conf_t *cf, ngx_command_t *cmd, void *conf);
    ngx_uint_t            conf;
    ngx_uint_t            offset;
    void                 *post;
};
```

我们重点关注这个type成员。有如下几个比较重要的type属性需要着重熟悉:

- NGX_MAIN_CONF
- NGX_DIRECT_CONF
- NGX_CONF_BLOCK

搜索一下nginx-1.6.2的源代码，发现，type值包含了NGX_DIRECT_CONF的指令所在的模块分别是:

- ngx_core_module模块的所有指令。
- ngx_openssl_module模块的所有指令。其实只有一个ssl_engine
- ngx_google_perftools_module模块的所有指令。其实只有一个google_perftools_profiles
- ngx_regex_module模块的所有指令。其实只有一个pcre_jit

这几个模块有如下共同点:

- 模块类型都是NGX_CORE_MODULE
- 指令类型NGX_DIRECT_CONF都是与NGX_MAIN_CONF共同出现

在Nginx中的几个基本模块和几个模块ctx类型中，我们讲过，Nginx-1.6.2中，ngx_core_module_t类型的模块（即ctx成员为ngx_core_module_t类型的ngx_module_t变量，也即type成员为NGX_CORE_MODULE的ngx_module_t变量)包括：

1. ngx_core_module
2. ngx_events_module

3. ngx_openssl_module
4. ngx_google_perftools_module
5. ngx_http_module
6. ngx_errlog_module
7. ngx_mail_module
8. ngx_regex_module

这些NGX_CORE_MODULE类型的模块中，还有如下几个，其配置指令集中没有NGX_DIRECT_CONF类型的指令:

1. ngx_events_module
2. ngx_http_module
3. ngx_errlog_module
4. ngx_mail_module

其中：

1. ngx_events_module模块只有一个指令"events",是一个配置块类型的指令，即指令类型包含NGX_CONF_BLOCK.
2. ngx_http_module模块只有一个指令"http",同样是一个配置块类型的指令,即指令类型包含NGX_CONF_BLOCK.
3. ngx_errlog_module
4. ngx_mail_module模块有两个指令“mail”和“imap”，且都是配置块类型的指令，即指令类型包括NGX_CONF_BLOCK.

搜索nginx-1.6.2代码，发现type值包含了NGX_MAIN_CONF的指令所在的模块为:

- ngx_core_module模块的所有指令。
- ngx_events_module模块的所有指令。其实只有一个“events”
- ngx_openssl_module模块的所有指令。其实只有一个“ssl_engine”
- ngx_google_perftools_module模块的所有指令。其实只有一个"google_perftools_profiles"
- ngx_http_module模块的所有指令，其实只有一个"http".
- ngx_errlog_module模块的所有指令，其实只有一个"error_log"
- ngx_mail_module模块的所有指令，包含"mail"和"imap"
- ngx_regex_module模块的所有指令，包含"pcre_jit"

这几个模块有如下共同点:

- 模块类型都是NGX_CORE_MODULE

所以，综上，在nginx-1.6.2中，关于NGX_MAIN_CONF和NGX_DIRECT_CONF有如下几点:

- NGX_MAIN_CONF和NGX_DIRECT_CONF都只会出现在NGX_CORE_MODULE类型的模块中。

- 所有的NGX_CORE_MODULE类型的模块，其指令均有NGX_MAIN_CONF属性
- 所有NGX_CORE_MODULE类型的模块，除了**ngx_errlog_module**模块外,其指令如果没有NGX_DIRECT_CONF属性，则会有NGX_CONF_BLOCK属性。ngx_errlog_module模块比较特殊，其指令属性为“NGX_MAIN_CONF|NGX_CONF_1MORE”
- 所有的nginx模块中，NGX_DIRECT_CONF和NGX_CONF_BLOCK属性不会同时出现。

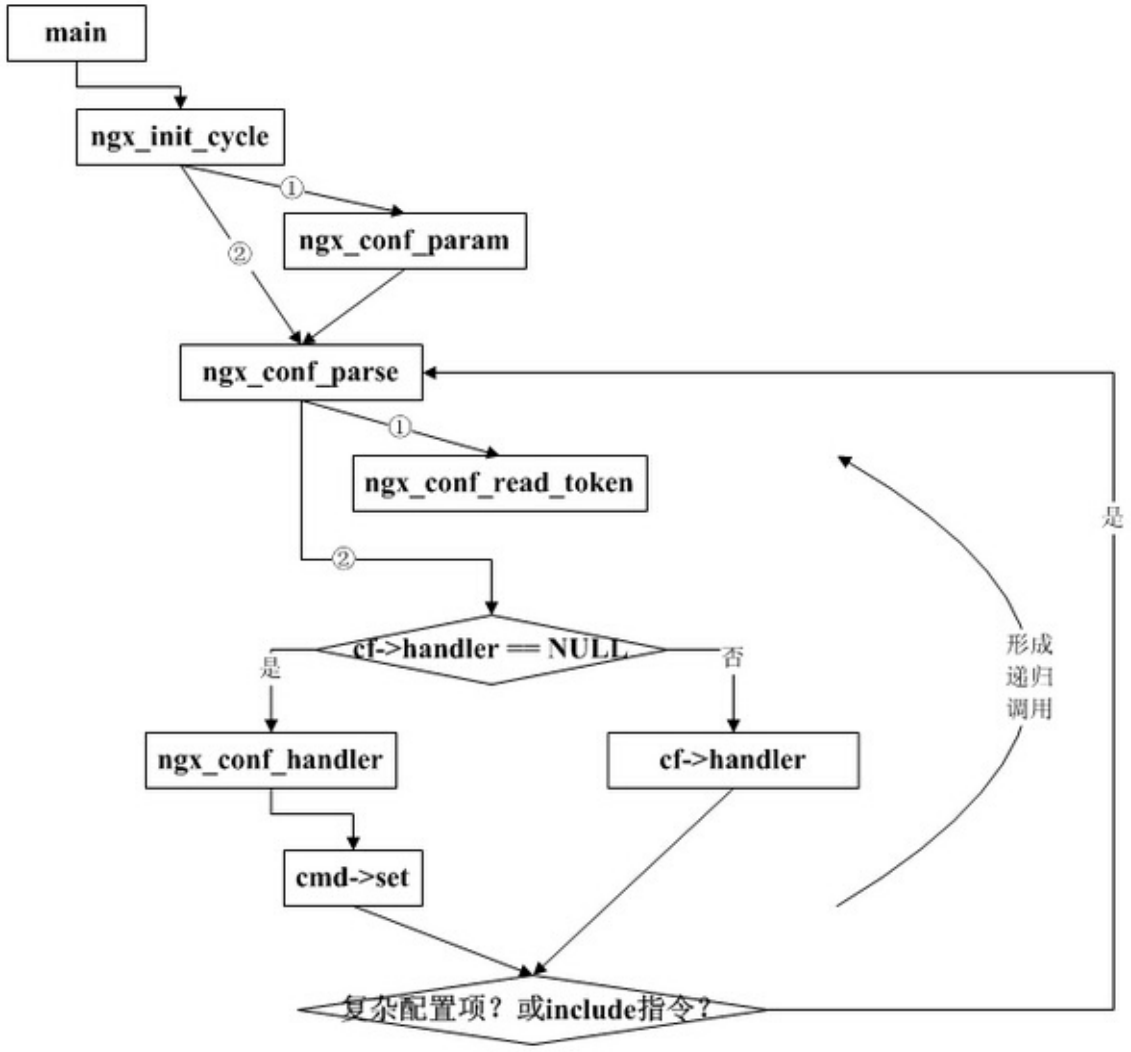
另外，type值包含了NGX_CONF_BLOCK属性的指令所在的模块有:

- ngx_events_module模块，指令只有“events”
- ngx_http_module模块，指令只有“http”
- ngx_http_charset_filter_module模块的“charset_map”指令
- ngx_http_core_module模块的server指令、location指令、types指令、limit_except指令
- ngx_http_geo_module模块的geo指令
- ngx_http_map_module模块的map指令
- ngx_http_rewrite_module模块的if指令
- ngx_http_split_clients_module模块的split_clients指令
- ngx_http_upstream_module模块的upstream指令
- ngx_mail_module模块的mail指令和imap指令
- ngx_mail_core_module模块的server指令

列出这些不是为了去死记硬背。而是我们可以很直观的了解到,NGX_CONF_BLOCK属性只能表示该命令后面跟的是一个复杂配置项而已。

Nginx配置解析的流程

Nginx配置解析的流程如下图所示:



参考文章http://tech.uc.cn/?p=300

nginx proxy_cache与etag配合的问题

首先谈谈遇到的问题:

一个Nginx架在一个后端服务的前面，Nginx proxy_pass到它并开启proxy_cache,假设这个后端服务总是会吐Etag响应头。 在应用中，我们发现当nginx 的proxy_cache成功将后端的页面cache住时,浏览器多次对该页面发起请求，会命中nginx的cache,但即使浏览器请求带了If-None-Match请求头，nginx却不会响应304，而是响应200. 这样带来的问题是，即使nginx的cache将请求阻挡在后端应用之外，但是: （1）命中后每次响应200导致我们nginx所在的服务器和客户浏览器双方都有流量损耗 （2）更重要的是增长了我们的服务响应时间。因为，如果是304的话，nginx不需要向浏览器吐数据，只用告诉浏览器用本地的缓存就好了。

排查过程涉及个版本的测试和代码比对，这里就省略了，下面说下结论:

一. **nginx-1.7.3**之前的版本:

- nginx很早就支持etag，但是nginx-1.7.3之前不支持弱etag。
- nginx-1.7.3之前版本在proxy_cache时，会有上面提到的问题。
- 之所以会遇到上面的问题，是因为nginx里面有很多filter模块，比如ssl,xsl,gzip等等，这些filter模块本质上算是对响应做出了修改。所以，Nginx为了严格遵循etag的本意，这种情况下就认为etag应该失效。而早期版本又不支持弱etag验证，所以干脆就不承认etag，每次都返回200,而不是304.

二.**nginx-1.7.3**及其之后的版本:

首先我们看看nginx-1.7.3的changelog：

```
Changes with nginx 1.7.3                                         08 Jul 2014
    *) Feature: weak entity tags are now preserved on response
       modifications, and strong ones are changed to weak.
    *) Feature: cache revalidation now uses If-None-Match header if
       possible.
    *) Feature: the "ssl_password_file" directive.
    *) Bugfix: the If-None-Match request header line was ignored if there
       was no Last-Modified header in a response returned from cache.
    *) Bugfix: "peer closed connection in SSL handshake" messages were
       logged at "info" level instead of "error" while connecting to
       backends.
    *) Bugfix: in the ngx_http_dav_module module in nginx/Windows.
    *) Bugfix: SPDY connections might be closed prematurely if caching was
       used.
```

可以看到，主要有两点：

- 增加了弱etag检验功能：对于那些修改了响应的filter模块，nginx启用弱etag检验。
- cache住的文件，其验证会优先采用If-None-Match校验，即Etag校验。

那么，在nginx-1.7.3及其以后的版本，其Etag功能可以描述如下:

1. Etag功能得到增强，既有强Etag，又有弱Etag.其实，但是所谓的强弱，只是为了遵循标准而分出来的两种说法而已。
2. 对于Nginx本地的静态文件，是强Etag验证。
3. 对于Nginx本地的非静态内容，不做Etag验证。这里可以用nginx-lua模块ngx.say来简单验证。
4. 对于proxy_cache缓存住的文件,无论该文件是后端的静态文件，或是后端动态产生的页面，只要后端吐出了Etag响应头，则Nginx对客户端过来的请求，都会启动Etag校验。即，第一次请求，Nginx会将后端吐出的Etag头传给客户端，客户端后面再请求时，会带上 If-None-Match请求头，如果校验通过，会直接返回304.（这就解决了我们的问题)
5. 注意，上面这一点又可以分为两种情况，第一种是如果Nginx在将cache住的内容吐给浏览器时，如果Nginx不启用filter模块来修改响应，则Etag的强弱跟后端传过来的相同。第二种情况，如果Nginx在将cache住的内容吐给浏览器时，如果启用了filter模块，即响应头或体被修改，那么Nginx会将后端的强Etag转换为弱Etag.例如：如果后端本来返回的Etag为ETag: "12345",则Nginx会将其弱化，吐给浏览器，即改为Etag: W/"12345", 浏览器下次请求的If-None-Match请求头也变为If-None-Match: W/"12345"。

不规范的Nginx开发

最近这一年多， 见识过很多基于 Nginx 开发的项目， 在这个过程中也遇到了几个很常见的不规范的做法。

离主分支过远

Nginx 是一个很有生命力的项目， 不断的在开发一些新的特性， 基于这种项目开发的项目，从一开始就要想到版本同步升级的问题。

Nginx 本身提供了强大的模块开发机制， 在做自己的业务开发时， 应该尽可能用模块去解决， 而不要乱动 Nginx 核心代码。其实， 如果不是业务场景特殊， 或者对性能有更苛刻的要求， 开发者都不应该去修改核心代码。 如果实在到了不动核心代码不行或者解决方案非常憋屈的时候， 也应该尽量先做好同步升级的方案， 比如经常不定期合入主干代码等。 连开发阵容强大的 Tengine, 都会跟进 Nginx 的更新， 你有什么理由不这样做呢。

这不， 我就见过一个基于 Nginx 的某个上古版本(0.7.x)开发的系统， 随着业务的积累， 已经将 Nginx 原始代码改的不像样子了， 甚至连基础的进程架构都改了。 但业务需求越来越多，这种方式明显跟不上节奏了， 一方面是新特性用不了， 另一方面是过多的改动导致潜在的 bug 很多。直到现在不得不痛下决心， 彻底更新 Nginx 版本， 去掉之前的那些核心改动。 虽然比较痛苦， 但方向是对的。当业界都在利用 Openresty 这一利器， 用 lua 开心的写着业务逻辑， 你还在一行一行的用 c 来写业务， 这项目怎么能维护得下去。

提到 Openresty, 不得不赞叹章宜春的版本策略。首先 Openresty 是一个bundle，其核心是 ngx_lua 模块， 这个模块本身是不会依赖对 Nginx 核心的改动。 而且 Openresty 跟进 Nginx 版本的速度也很快， 当前 Openresty 的 release 已经到 Nginx-1.9.3 了。

真的是一定要动核心吗？

事实上，并不是。 很多时候， 所谓 “不得不” 进行的对核心的修改， 只是因为开发者对 Nginx 本身理解的不够透彻。

例如， 一个简单的场景： 一个模块 A 需要为当前请求生成几个内部变量， 既然是与当前请求相关的， 很自然的会想到在 ngx_http_request_s 结构体里增加一个成员。 可是， Nginx 已经提供了请求上下文的的 ctx 机制了， 使用起来干净、简单。

再例如， Nginx 的各模块之间本应互相独立的。 但出于代码复用等目的， 实际项目里会出现模块之间互相依赖的情况。 比方说， A 模块固定完成一个功能， 所有其他的业务模块， 都去操作当前请求的对应 A 模块的上下文 ctx， 而 A 模块只去根据 ctx 做相应的处理。 这种做

法看上去很精明， 却毫不优雅。 导致了模块之间的耦合， 编译 B 模块就必须把 A 模块一起编译进来。 其实深入思考一下， 这些都可以用很好的方式来实现。

末尾

这篇文章似乎是在吐槽一些开发者， 但实际不是得。 从公司的角度， 不管使用什么办法，能撑起业务来， 就是英雄， 相应的， 开发周期的制定里， 并不会去给那么多时间让开发者去透彻理解一个系统后再开始开发。

所以，当我们看到这些 “老旧” 的系统， 虽然他们可能很难维护， 但我们还是应该对当初开发的人持有敬意。

Nginx对Connection头的处理过程

1. 标准

RFC2616 中，对 Connection 的说明如下：

> HTTP/1.1 proxies MUST parse the Connection header field before a message is forwarded and, for each connection-token in this field, remove any header field(s) from the message with the same name as the connection-token. Connection options are signaled by the presence of a connection-token in the Connection header field, not by any corresponding additional header field(s), since the additional header field may not be sent if there are no parameters associated with that connection option.

综合RFC2626 14.10、《HTTP权威指南》4.3.1、《图解HTTP》6.3.2中的说法，均指明了 Connection 头部（请求头、响应头）主要包括如下两方面的作用：

1. 控制不再转发给代理的首部字段
2. 管理持久连接

其中，我个人经常见到的是第二种用法，对第一种用法还不甚了解。

第一种用法，大概意思就是，在一个 HTTP 应用（客户端、服务器）将报文转发出之前，必须删除 Connection 首部列出的所有首部字段（多个不同的字段用逗号分隔），当然一些 end-to-end 的头部是肯定不能放进去的，例如 Cache-Control 头。

2.Nginx 是如何做的

注:以下代码均来自 Nginx-1.8.0

对于请求中的 Connection 头， Nginx 在 http 解析时调用 ngx_http_process_connection 方法：

```
static ngx_int_t
ngx_http_process_connection(ngx_http_request_t *r, ngx_table_elt_t *h,
    ngx_uint_t offset)
{
    if (ngx_strcasestrn(h->value.data, "close", 5 - 1)) {
        r->headers_in.connection_type = NGX_HTTP_CONNECTION_CLOSE;

    } else if (ngx_strcasestrn(h->value.data, "keep-alive", 10 - 1)) {
        r->headers_in.connection_type = NGX_HTTP_CONNECTION_KEEP_ALIVE;
    }

    return NGX_OK;
}
```

可见，该方法主要功能只涉及到上面所述的 Connection 作用的第2个作用，而没有第1个。这应该算的上是实现对标准的支持不完整吧。

不管怎么样，我们继续分析一下 Nginx 在管理持久连接上具体是怎么做的，上面的代码逻辑很简单：（1）如果 Connection 头是 close（不区分大小写），则 r->headers_in.connection_type 被置为 NGX_HTTP_CONNECTION_CLOSE （2）如果 Connection 头是 keep-alive（不区分大小写），则 r->headers_in.connection_type 被设置为 NGX_HTTP_CONNECTION_KEEP_ALIVE （3）如果不是以上两种情况，则什么都不做，此时 r->headers_in.connection_type 默认为0.

可以看到，ngx_http_process_connection 的作用仅仅是设置了一个标记r->headers_in.connection_type，我们继续看这个标记是如何被使用的。

要想完整的讲述这个过程，不可避免需要涉及一些 Nginx 配置解析、解析 HTTP 协议相关的流程，但这些都不是本文的重点，下面均一笔带过。

Nginx 的各历史版本中，对于 http 协议解析的过程，细节稍微有些变化。在 Nginx-1.8.0 中，过程如下：

（0）我们知道，Nginx 是一个主进程加多子进程的架构，配置解析发生在 fork 子进程之前，在这个过程中很多操作都是对于全局变量 cycle 的操作。在 fork 之后，每个子进程均会继承一份这个全局变量 cycle(这里不考虑COW)。

（1）http 配置解析的入口为 ngx_http_block，这是个很复杂的函数。因为 Nginx 的配置文件分层级，可以有包含关系，为了对这个特性提供支持，配置文件的解析涉及到配置项的内存布局的设计，最终落实下来，就是全局变量 cycle 的 conf_ctx 成员，这是个四重指针。配置解析的这部分是另一个话题，本文不打算细说。这里要关注的是，在 ngx_http_block 函数的最后，调用了ngx_http_optimize_servers 方法，在这个方法里，完成了这样一个事情：配置文件里的监听套接字（可能是多个），最终被复制到了全局变量 ngx_cycle 的 listening 数组

里。整个调用关系为，nginx.c->main()->ngx_init_cycle->ngx_http_block(在 ngx_init_cycle里通过钩子被回调)->ngx_http_optimize_servers->ngx_http_init_listening->ngx_http_add_listening->ngx_create_listening, 有兴趣的同学可以深入进去看看。

（2）在上面的这个调用链里，ngx_http_add_listening 做了另外一个事情：将所有的 ngx_listening_t 类型的监听套接字的 handler 钩子设置为 ngx_http_init_connection，这个在后面会用到。

（3）nginx fork 出多个子进程，每个子进程会在 ngx_worker_process_init 方法里调用各个 nginx 模块 init_process 钩子，其中当然也包括 NGX_EVENT_MODULE 类型的 ngx_event_core_module 模块，其 init_process 钩子为 ngx_event_process_init。在 ngx_event_process_init 里，每一个 ngx_listening_t 类型的监听套接字变量 ls[i]，根据 ngx_get_connection 从 nginx 的 connections 储备池里获得一个与之相关的ngx_connection_t 类型的变量 c ,这两个变量均有一个指针成员指向对方，以此保持互相联系。从这里开始，我们将注意力转移到这个 ngx_connection_t 类型的变量 c 上。在 ngx_event_process_init 的后面，这个 ngx_connection_t 类型的变量的读事件的 handler，被置为 ngx_event_accept。然后这个读事件被添加到 epoll 中。

（4）当一个请求来临时，ngx_event_accpet 被回调，其中上面第（2）步里为监听套接字设置的 handler,即 ngx_http_init_connection 被调用：

```
ls->handler(c);
```

这个调用非常有趣，ls 实质上是代表着监听套接字，而参数 c 则是 accept 后建立起来的连接套接字，根据 socket 的基本知识，该连接上后续客户端与 Nginx 之间的信息传输，都通过这个连接套接字上的读写来进行。

（5）从 ngx_http_init_connection 开始，就着手进行一系列 HTTP 协议解析。中间涉及到 ngx_http_wait_request_handler、ngx_http_create_request、ngx_http_process_request_line、ngx_http_process_request_headers 等解析方法。

似乎已经偏题太远了，我们回到最初的 Connection 请求头，在上面所讲的 ngx_http_process_connection 中，根据 Connection 头，将 r->headers_in.connection_type 置为NGX_HTTP_CONNECTION_CLOSE、NGX_HTTP_CONNECTION_KEEP_ALIVE 或者默认初始值 0.

那么这个过程发生在上面所述的一系列流程的哪个阶段呢？当然是发生在 ngx_http_process_request_headers 里了，在个方法里，全局数组 ngx_http_headers_in 的对应元素的 handler 被调用，对于 Connection 请求头，就是 ngx_http_process_connection 了。

讲明白了 ngx_http_process_connection 发生的前因，我们再来看看后果。

ngx_http_process_connection 的直接影响只有一个，即对 r->headers_in.connection_type进行赋值，close 为 NGX_HTTP_CONNECTION_CLOSE，keep-alive 为 NGX_HTTP_CONNECTION_KEEP_ALIVE，其它情况为 0（初始值）。

（6）在第（5）步里，在 ngx_http_process_request_line方法调用完 ngx_http_process_request_headers 后，继续调用 ngx_http_process_request，进而开始正式的在业务上处理HTTP请求。ngx_http_process_request的核心内容是对 ngx_http_handler 的调用。

（7）在 ngx_http_handler 里，有这样一段代码：

```
if (!r->internal) {
        switch (r->headers_in.connection_type) {
        case 0:
            r->keepalive = (r->http_version > NGX_HTTP_VERSION_10);
            break;

        case NGX_HTTP_CONNECTION_CLOSE:
            r->keepalive = 0;
            break;

        case NGX_HTTP_CONNECTION_KEEP_ALIVE:
            r->keepalive = 1;
            break;
        }

        r->lingering_close = (r->headers_in.content_length_n > 0
                              || r->headers_in.chunked);
        r->phase_handler = 0;

    } else {
        cmcf = ngx_http_get_module_main_conf(r, ngx_http_core_module);
        r->phase_handler = cmcf->phase_engine.server_rewrite_index;
    }
```

所以，到现在为止，客户端请求里带来的 Connection 头部，落在了 r->keepalive 上了。规则如下：

- 如果 Connection 头部里为"close",则 r->keepalive 为 0.
- 如果 Connection 头部里为"keep-alive",则 r->keepalive为 1.
- 如果不是以上两种情况，则按照 HTTP 协议走默认情况：如果是 HTTP 1.0 以上，则 r->keepalive 默认为 1，如果是 HTTP 1.0及以下，则 r->keepalive默认为 0.这一点是与 HTTP 协议相符合的.

到目前为止，我们所讲的都算是 r->keepalive 是怎么产生的，还没有涉及到它是如何被使用的。

r->keepalive 的使用主要是在函数 ngx_http_finalize_connection 中，而 ngx_http_finalize_connection 在 Nginx 中，仅被 ngx_http_finalize_request 调用。顾名思义，ngx_http_finalize_request 讲的是怎么结束一个 HTTP 请求的。

（8）在 ngx_http_finalize_connection 中，如果 r->keepalive 为1，则会调用 ngx_http_set_keepalive 并返回。ngx_http_set_keepalive 方法完成将当前连接设为keepalive状态的实质性工作。它实际上会把表示请求的 ngx_http_request_t 结构体释放，却又不会调用 ngx_http_close_connection 方法关闭连接，同时也在检测 keepalive 连接是否超时。

至此，Connection 头部在 Nginx 里的处理流程差不多都讲完了，具体细节可按照这个脉络查看相应的代码。

Nginx-1.9.8推出的切片模块

熟悉 CDN 行业主流技术的朋友应该都比较清楚，虽然 Nginx 近几年发展的如日中天，但是基本上没有直接使用它自带的 proxy_cache 模块来做缓存的，原因有很多，例如下面几个：

- 不支持多盘
- 不支持裸设备
- 大文件不会切片
- 大文件的 Range 请求表现不尽如人意
- Nginx 自身不支持合并回源

当然，上面列出来的并不全， 所以，在现在主流的 CDN 技术栈里面， Nginx 起到的多是一个粘合剂的作用，例如调度器、负载均衡器、业务逻辑（防盗链等），需要与 Squid、ATS 等主流 Cache Server 配合使用，即使不使用这些技术，也会使用其它办法，例如直接使用文件系统和数据库去管理文件，而不会直接使用 proxy_cache 模块。

Nginx-1.9.8 中新增加的一个模块ngx_http_slice_module解决了一部分问题。本文就来尝尝鲜，看看这个切片模块。

注意：截至到发文时，Nginx 马上发布了 Nginx-1.9.9，用来解决 Nginx-1.9.8中的一个 Bug，所以，在实际使用中，如果需要使用本新增特性，请直接使用 Nginx-1.9.9。

首先，我们看看几个不同版本的 Nginx 的 proxy_cache 对 Range 的处理情况。

Nginx-0.8.15

在 Nginx-0.8.15 中，使用如下配置文件做测试:

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    proxy_cache_path /tmp/nginx/cache levels=1:2 keys_zone=cache:100m;
    server {
        listen       8087;
        server_name  localhost;
        location / {
            proxy_cache cache;
            proxy_cache_valid 200 206 1h;
           # proxy_set_header Range $http_range;
            proxy_pass http://127.0.0.1:8080;

        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }

    }
}
```

重点说明以下两种情况：

- 第一次 Range 请求（没有本地缓存），Nginx 会去后端将整个文件拉取下来（后端响应码是200）后，并且返回给客户端的是整个文件，响应状态码是200，而非206. 后续的 Range 请求则都用缓存下来的本地文件提供服务，且响应状态码都是对应的206了。
- 如果在上面的配置文件中，加上 proxy_set_header Range $http_range;再进行测试(测试前先清空 Nginx 本地缓存)。则第一次 Range 请求（没有本地缓存），Nginx 会去后端用 Range 请求文件，而不会把整个文件拉下来，响应给客户端的也是206.但问题在于，由于没有把 Range 请求加入到 cache key 中，会导致后续所有的请求，不管 Range 如何，只要 url 不变，都会直接用cache 的内容来返回给客户端，这肯定是不符合要求的。

Nginx-1.9.7

在 Nginx-1.9.7 中，同样进行上面两种情况的测试，第二种情况的结果其实是没多少意义，而且肯定也和 Nginx-0.8.15 一样，所以这里只关注第一种测试情况。

第一次 Range 请求（没有本地缓存），Nginx 会去后端将整个文件拉取下来（后端响应码是200），但返回给客户端的是正确的 Range 响应，即206.后续的 Range 请求，则都用缓存下来的本地文件提供服务，且都是正常的206响应。

可见，与之前的版本相比，还是有改进的，但并没有解决最实质的问题。

我们可以看看 Nginx 官方对于 Cache 在 Range 请求时行为的说明:

> How Does NGINX Handle Byte Range Requests?
>
> If the file is up-to-date in the cache, then NGINX honors a byte range request and serves only the specified bytes of the item to the client. If the file is not cached, or if it's stale, NGINX downloads the entire file from the origin server. If the request is for a single byte range, NGINX sends that range to the client as soon as it is encountered in the download stream. If the request specifies multiple byte ranges within the same file, NGINX delivers the entire file to the client when the download completes.
>
> Once the download completes, NGINX moves the entire resource into the cache so that all future byte-range requests, whether for a single range or multiple ranges, are satisfied immediately from the cache.

Nginx-1.9.8

我们继续看看Nginx-1.9.8, 当然，在编译时要加上参数--with-http_slice_module，并作类似下面的配置:

```
http {
    include       mime.types;
    default_type  application/octet-stream;
    sendfile        on;
    keepalive_timeout  65;

    proxy_cache_path /tmp/nginx/cache levels=1:2 keys_zone=cache:100m;
    server {
        listen       8087;
        server_name  localhost;
        location / {
            slice 1m;
            proxy_cache cache;
            proxy_cache_key $uri$is_args$args$slice_range;
            proxy_set_header Range $slice_range;
            proxy_cache_valid 200 206 1h;
            #proxy_set_header Range $http_range;
            proxy_pass http://127.0.0.1:8080;

        }
        error_page   500 502 503 504  /50x.html;
        location = /50x.html {
            root   html;
        }

    }
}
```

不测不知道，一侧吓一跳，这俨然是一个杀手级的特性。

首先，如果不带 Range 请求，后端大文件在本地 cache 时，会按照配置的 slice 大小进行切片存储。

其次，如果带 Range 请求，则 Nginx 会用合适的 Range 大小（以 slice 为边界）去后端请求，这个大小跟客户端请求的 Range 可能不一样，并将以 slice 为大小的切片存储到本地，并以正确的206响应客户端。

注意上面所说的，Nginx 到后端的 Range 并不一定等于客户端请求的 Range，因为无论你请求的Range 如何，Nginx 到后端总是以 slice 大小为边界，将客户端请求分割成若干个子请求到后端，假设配置的 slice 大小为1M,即1024字节，那么如果客户端请求 Range 为0-1023范围以内任何数字，均会落到第一个切片上，如果请求的 Range 横跨了几个 slice 大小，则 nginx会向后端发起多个子请求，将这几个 slice 缓存下来。而对客户端，均以客户端请求的 Range 为准。如果一个请求中，有一部分文件之前没有缓存下来，则 Nginx 只会去向后端请求缺失的那些切片。

由于这个模块是建立在子请求的基础上，会有这么一个潜在问题：当文件很大或者 slice 很小的时候，会按照 slice 大小分成很多个子请求，而这些个子请求并不会马上释放自己的资源，可能会导致文件描述符耗尽等情况。

小结

总结一下，需要注意的点：

- 该模块用在 proxy_cache 大文件的场景，将大文件切片缓存
- 编译时对 configure 加上 --with-http_slice_module 参数
- $slice_range 一定要加到 proxy_cache_key 中，并使用 proxy_set_header 将其作为 Range 头传递给后端
- 要根据文件大小合理设置 slice 大小

具体特性的说明，可以参考 Roman Arutyunyan 提出这个 patch 时的邮件来往：
https://forum.nginx.org/read.php?29,261929,261929#msg-261929

顺带提一下，Roman Arutyunyan 也是个大牛，做流媒体领域的同学们肯定很多都听说过：nginx-rtmp 模块的作者。

参考资料

1. Nginx 官方的 Cache 指南 https://www.nginx.com/blog/nginx-caching-guide/
2. Nginx各版本changelog http://nginx.org/en/CHANGES
3. Nginx proxy 模块 wiki http://nginx.org/en/docs/http/ngx_http_proxy_module.html
4. http_slice_module 的历次提交记录 http://hg.nginx.org/nginx/rev/29f35e60840b http://hg.nginx.org/nginx/rev/bc9ea464e354 http://hg.nginx.org/nginx/rev/4f0f4f02c98f

5. http_slice_module 提交前的邮件来往 https://forum.nginx.org/read.php?29,261929
6. Nginx 之前版本关于 Range cache 的邮件来往 https://forum.nginx.org/read.php?2,8958,8958
7. 切片模块的 wiki http://nginx.org/en/docs/http/ngx_http_slice_module.html

1. 什么叫惊群现象

首先，我们看看维基百科对惊群的定义：

> The thundering herd problem occurs when a large number of processes waiting for an event are awoken when that event occurs, but only one process is able to proceed at a time. After the processes wake up, they all demand the resource and a decision must be made as to which process can continue. After the decision is made, the remaining processes are put back to sleep, only to all wake up again to request access to the resource.
>
> This occurs repeatedly, until there are no more processes to be woken up. Because all the processes use system resources upon waking, it is more efficient if only one process was woken up at a time.
>
> This may render the computer unusable, but it can also be used as a technique if there is no other way to decide which process should continue (for example when programming with semaphores).

简而言之，惊群现象（thundering herd）就是当多个进程和线程在同时阻塞等待同一个事件时，如果这个事件发生，会唤醒所有的进程，但最终只可能有一个进程/线程对该事件进行处理，其他进程/线程会在失败后重新休眠，这种性能浪费就是惊群。

2. accept 惊群

考虑如下场景：主进程创建socket, bind, listen之后，fork出多个子进程，每个子进程都开始循环处理（accept)这个socket。每个进程都阻塞在accpet上，当一个新的连接到来时，所有的进程都会被唤醒，但其中只有一个进程会accept成功，其余皆失败，重新休眠。这就是accept惊群。

那么这个问题真的存在吗？

事实上，历史上，Linux 的 accpet 确实存在惊群问题，但现在的内核都解决该问题了。即，当多个进程/线程都阻塞在对同一个 socket 的 accept 调用上时，当有一个新的连接到来，内核只会唤醒一个进程，其他进程保持休眠，压根就不会被唤醒。

测试代码如下：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <stdio.h>
#include <string.h>
#define PROCESS_NUM 10
int main()
{
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    int connfd;
    int pid;
    char sendbuff[1024];
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(1234);
    bind(fd, (struct sockaddr*)&serveraddr, sizeof(serveraddr));
    listen(fd, 1024);
    int i;
    for(i = 0; i < PROCESS_NUM; i++)
    {
        int pid = fork();
        if(pid == 0)
        {
            while(1)
            {
                connfd = accept(fd, (struct sockaddr*)NULL, NULL);
                snprintf(sendbuff, sizeof(sendbuff), "accept PID is %d\n", getpid());

                send(connfd, sendbuff, strlen(sendbuff) + 1, 0);
                printf("process %d accept success!\n", getpid());
                close(connfd);
            }
        }
    }
    int status;
    wait(&status);
    return 0;
}
```

当我们对该服务器发起连接请求（用 telnet/curl 等模拟）时，会看到只有一个进程被唤醒。

关于 accept 惊群的一些帖子或文章：

- 惊群问题在 linux 上可能是莫须有的问题
- Does the Thundering Herd Problem exist on Linux anymore?
- 历史上解决 linux accept 惊群的补丁讨论

3. epoll惊群

如上所述，accept 已经不存在惊群问题，但 epoll 上还是存在惊群问题。即，如果多个进程/线程阻塞在监听同一个 listening socket fd 的 epoll_wait 上，当有一个新的连接到来时，所有的进程都会被唤醒。

考虑如下场景：

主进程创建 socket, bind， listen 后，将该 socket 加入到 epoll 中，然后 fork 出多个子进程，每个进程都阻塞在 epoll_wait 上，如果有事件到来，则判断该事件是否是该 socket 上的事件，如果是，说明有新的连接到来了，则进行 accept 操作。为了简化处理，忽略后续的读写以及对 accept 返回的新的套接字的处理，直接断开连接。

那么，当新的连接到来时，是否每个阻塞在 epoll_wait 上的进程都会被唤醒呢？

很多博客中提到，测试表明虽然 epoll_wait 不会像 accept 那样只唤醒一个进程/线程，但也不会把所有的进程/线程都唤醒。例如这篇文章：关于多进程 epoll 与 “惊群”问题。

为了验证这个问题，我自己写了一个测试程序：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <netdb.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/wait.h>
#define PROCESS_NUM 10
static int
create_and_bind (char *port)
{
    int fd = socket(PF_INET, SOCK_STREAM, 0);
    struct sockaddr_in serveraddr;
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
    serveraddr.sin_port = htons(atoi(port));
    bind(fd, (struct sockaddr*)&serveraddr, sizeof(serveraddr));
    return fd;
}
    static int
make_socket_non_blocking (int sfd)
{
    int flags, s;

    flags = fcntl (sfd, F_GETFL, 0);
```

```
    if (flags == -1)
    {
        perror ("fcntl");
        return -1;
    }

    flags |= O_NONBLOCK;
    s = fcntl (sfd, F_SETFL, flags);
    if (s == -1)
    {
        perror ("fcntl");
        return -1;
    }

    return 0;
}

#define MAXEVENTS 64

int
main (int argc, char *argv[])
{
    int sfd, s;
    int efd;
    struct epoll_event event;
    struct epoll_event *events;

    sfd = create_and_bind("1234");
    if (sfd == -1)
        abort ();

    s = make_socket_non_blocking (sfd);
    if (s == -1)
        abort ();

    s = listen(sfd, SOMAXCONN);
    if (s == -1)
    {
        perror ("listen");
        abort ();
    }

    efd = epoll_create(MAXEVENTS);
    if (efd == -1)
    {
        perror("epoll_create");
        abort();
    }

    event.data.fd = sfd;
    //event.events = EPOLLIN | EPOLLET;
    event.events = EPOLLIN;
    s = epoll_ctl(efd, EPOLL_CTL_ADD, sfd, &event);
```

```
    if (s == -1)
    {
        perror("epoll_ctl");
        abort();
    }

    /* Buffer where events are returned */
    events = calloc(MAXEVENTS, sizeof event);
            int k;
    for(k = 0; k < PROCESS_NUM; k++)
    {
        int pid = fork();
        if(pid == 0)
        {

            /* The event loop */
            while (1)
            {
                int n, i;
                n = epoll_wait(efd, events, MAXEVENTS, -1);
                printf("process %d return from epoll_wait!\n", getpid());
                                    /* sleep here is very important!*/
                //sleep(2);
                                    for (i = 0; i < n; i++)
                {
                    if ((events[i].events & EPOLLERR) || (events[i].events & EPOLLHUP) ||
                    {
                        /* An error has occured on this fd, or the socket is not
                        ready for reading (why were we notified then?) */
                        fprintf (stderr, "epoll error\n");
                        close (events[i].data.fd);
                        continue;
                    }
                    else if (sfd == events[i].data.fd)
                    {
                        /* We have a notification on the listening socket, which
                        means one or more incoming connections. */
                        struct sockaddr in_addr;
                        socklen_t in_len;
                        int infd;
                        char hbuf[NI_MAXHOST], sbuf[NI_MAXSERV];

                        in_len = sizeof in_addr;
                        infd = accept(sfd, &in_addr, &in_len);
                        if (infd == -1)
                        {
                            printf("process %d accept failed!\n", getpid());
                            break;
                        }
                        printf("process %d accept successed!\n", getpid());

                        /* Make the incoming socket non-blocking and add it to the
                        list of fds to monitor. */
```

```
                        close(infd);
                    }
                }
            }
        }
    }
    int status;
    wait(&status);
    free (events);
    close (sfd);
    return EXIT_SUCCESS;
}
```

发现确实如上面那篇博客里所说，当我模拟发起一个请求时，只有一个或少数几个进程被唤醒了。

也就是说，到目前为止，还没有得到一个确定的答案。但后来，在下面这篇博客中看到这样一个评论：http://blog.csdn.net/spch2008/article/details/18301357

> 这个总结，需要进一步阐述，你的实验，看上去是只有4个进程唤醒了，而事实上，其余进程没有被唤醒的原因是你的某个进程已经处理完这个 accept，内核队列上已经没有这个事件，无需唤醒其他进程。你可以在 epoll 获知这个 accept 事件的时候，不要立即去处理，而是 sleep 下，这样所有的进程都会被唤起

看到这个评论后，我顿时如醍醐灌顶，重新修改了上面的测试程序，即在 epoll_wait 返回后，加了个 sleep 语句，这时再测试，果然发现所有的进程都被唤醒了。

所以，epoll_wait上的惊群确实是存在的。

4. 为什么内核不处理 epoll 惊群

看到这里，我们可能有疑惑了，为什么内核对 accept 的惊群做了处理，而现在仍然存在 epoll 的惊群现象呢？

我想，应该是这样的：accept 确实应该只能被一个进程调用成功，内核很清楚这一点。但 epoll 不一样，他监听的文件描述符，除了可能后续被 accept 调用外，还有可能是其他网络 IO 事件的，而其他 IO 事件是否只能由一个进程处理，是不一定的，内核不能保证这一点，这是一个由用户决定的事情，例如可能一个文件会由多个进程来读写。所以，对 epoll 的惊群，内核则不予处理。

5. Nginx 是如何处理惊群问题的

在思考这个问题之前，我们应该以前对前面所讲几点有所了解，即先弄清楚问题的背景，并能自己复现出来，而不仅仅只是看书或博客，然后再来看看 Nginx 的解决之道。这个顺序不应该颠倒。

首先，我们先大概梳理一下 Nginx 的网络架构，几个关键步骤为：

1. Nginx 主进程解析配置文件，根据 listen 指令，将监听套接字初始化到全局变量 ngx_cycle 的 listening 数组之中。此时，监听套接字的创建、绑定工作早已完成。
2. Nginx 主进程 fork 出多个子进程。
3. 每个子进程在 ngx_worker_process_init 方法里依次调用各个 Nginx 模块的 init_process 钩子，其中当然也包括 NGX_EVENT_MODULE 类型的 ngx_event_core_module 模块，其 init_process 钩子为 ngx_event_process_init。
4. ngx_event_process_init 函数会初始化 Nginx 内部的连接池，并把 ngx_cycle 里的监听套接字数组通过连接池来获得相应的表示连接的 ngx_connection_t 数据结构，这里关于 Nginx 的连接池先略过。我们主要看 ngx_event_process_init 函数所做的另一个工作：如果在配置文件里没有开启accept_mutex锁，就通过 ngx_add_event 将所有的监听套接字添加到 epoll 中。
5. 每一个 Nginx 子进程在执行完 ngx_worker_process_init 后，会在一个死循环中执行 ngx_process_events_and_timers，这就进入到时间处理的核心逻辑了。
6. 在 ngx_process_events_and_timers 中，如果在配置文件里开启了 accept_mutext 锁，子进程就会去获取 accet_mutext 锁。如果获取成功，则通过 ngx_enable_accept_events 将监听套接字添加到 epoll 中，否则，不会将监听套接字添加到 epoll 中，甚至有可能会调用 ngx_disable_accept_events 将监听套接字从 epoll 中删除（如果在之前的连接中，本worker子进程已经获得过accept_mutex锁）。
7. ngx_process_events_and_timers 继续调用 ngx_process_events，在这个函数里面阻塞调用 epoll_wait。

至此，关于 Nginx 如何处理 fork 后的监听套接字，我们已经差不多理清楚了，当然还有一些细节略过了，比如在每个 Nginx 在获取 accept_mutex 锁前，还会根据当前负载来判断是否参与 accept_mutex 锁的争夺。

把这个过程理清了之后，Nginx 解决惊群问题的方法也就出来了，就是利用 accept_mutex 这把锁。

如果配置文件中没有开启 accept_mutex，则所有的监听套接字不管三七二十一，都加入到 epoll中，这样当一个新的连接来到时，所有的 worker 子进程都会惊醒。

如果配置文件中开启了 accept_mutex，则只有一个子进程会将监听套接字添加到 epoll 中，这样当一个新的连接来到时，当然就只有一个 worker 子进程会被唤醒了。

6. 小结

现在我们对惊群及 Nginx 的处理总结如下：

- accept 不会有惊群，epoll_wait 才会。
- Nginx 的 accept_mutex,并不是解决 accept 惊群问题，而是解决 epoll_wait 惊群问题。
- 说Nginx 解决了 epoll_wait 惊群问题，也是不对的，它只是控制是否将监听套接字加入到 epoll 中。监听套接字只在一个子进程的 epoll 中，当新的连接来到时，其他子进程当然不会惊醒了。

7. 其他参考文章

“惊群”，看看 nginx 是怎么解决它的

系统编程

O_EXCL的作用

1.原始语义

与O_CREATE标志组合起来调用open，确保指定的文件由open的调用者创建，否则返回错误。即，如果进程A用O_CREATE和O_EXCL标志来调用open，期望创建一个指定的文件file1,如果file1不存在，则open成功返回且创建file1，如果file1已经存在了（即不是由进程A创建的），那么open返回错误。

2.使用场景

O_CREATE|O_EXCL多用于确保一个一个程序只能执行单个进程，不能执行多个进程。原理如下，假设进程A是某程序的一个实例，如果它用O_CREATE|O_EXCL标志能够成功创建指定的文件，说明它是该程序的唯一实例，可以继续执行；如果返回错误，说明该文件已经存在，进而说明系统中已经运行着一个该程序的其它实例，检测到错误的返回值后，该实例就可以退出了。

之所以能这么用的唯一理由是该操作是原子的。

之所以这么说，理由如下。假设同样语义的非原子的操作流程如下:

```
if( access(file, R_OK) == -1 )   /* 首先检查文件是否存在 */
    open(file, O_RDWR | O_CREAT, 0666);  /* 如果不存在，那我创建一个这样的文件 */
...  /* 继续执行任务 */
```

由于判断文件是否存在与创建文件是两个步骤，就会存在临界竞争的问题。试想下面的场景：

1. 某程序的进程A判断文件不存在，因此A认为自己是此时系统中该程序唯一的实例，准备继续执行创建指定的文件。
2. 操作系统的调度策略恰好在此时起作用，进程A暂停执行。此时指定的文件还没有创建。
3. 该程序的另一个进程B开始执行，同样，由于指定的文件不存在，B也认为自己是此时系统中该程序的唯一实例，准备继续执行创建指定的文件。
4. 这样，进程A和进程B都能成功调用open并继续往下执行。
5. 此时系统中就同时运行着该程序的两个实例，与仅运行一个进程的期望不符。

所以说，O_EXCL与O_CREATE联合使用的前提就是该操作是原子的。

3.非原子操作如何达到同样目的

假设现在不能使用O_EXCL|O_CREATE，或者假设用O_EXCL|O_CREATE调用open并不是原子的，该如何达到上面关于“一个程序只能运行一个实例”的要求呢？

可以用系统调用link来实现。

link的原型如下:

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

link的作用是为oldpath指定的源文件创建一个newpath指定的链接文件(硬链接，hard link)。如果创建成功，则返回0，如果newpath路径指定的目标文件在调用link前已经存在，则link会错误返回。

根据link的特点，可以达到上面的要求：

1. 首先确保文件系统中已经有一个源文件file1。
2. 某程序的一个进程A开始执行，调用link，试图创建一个file1的链接文件file2。
3. 如果A调用link成功，说明该程序此时只有进程A锁定了该文件，进程A可以继续往下执行。
4. 由于进程A为file1创建了一个链接文件file2,此时file1的链接数是2（用stat可获取链接数）。
5. 进程B调用link,同样试图创建file1的链接文件file2，但由于file2已经存在，link错误返回。可进一步调用stat系统调用，查看file1的链接数，确定该链接数是2。
6. 进程B退出。

4. O_EXCL|O_CREATE确实有可能是非原子的

在NFS上，O_EXCL|O_CREATE确实有可能是非原子的：

> On NFS, O_EXCL is supported only when using NFSv3 or later on kernel 2.6 or later. In NFS environments where O_EXCL support is not provided, programs that rely on it for performing locking tasks will contain a race condition.

在这种情况下，上面讨论的使用link的方法就有了用武之地。如上引文所述，在最新的内核中，NFS中并不存在该问题，用O_EXCL|O_CREATE仍然能满足要求。

5.其它

O_EXCL一般只能和O_CREATE一起使用，不能单独使用，但有一个例外：在2.6即以后的内核中，如果open指定的文件是一个块设备文件，O_EXCL可以单独使用，此时，如果该块设备正在被使用（例如已经被挂在)，那么open将失败返回，错误码是EBUSY;除此之外，单独使用O_EXCL的后果无法预知：

> In general, the behavior of O_EXCL is undefined if it is used without O_CREAT. There is one exception: on Linux 2.6 and later, O_EXCL can be used without O_CREAT if pathname refers to a block device. If the block device is in use by the system (e.g., mounted), open() fails with the error EBUSY.

vfork 挂掉的一个问题

注意：本文并非原创，转自陈皓的博客:**vfork挂掉的一个问题**，版权归陈皓所有。

在知乎上，有个人问了这样的一个问题——为什么vfork的子进程里用return，整个程序会挂掉，而且exit()不会？并给出了如下的代码，下面的代码一运行就挂掉了，但如果把子进程的return改成exit(0)就没事。

我受邀后本来不想回答这个问题的，因为这个问题明显就是RTFM的事，后来，发现这个问题放在那里好长时间，而挂在下面的几个答案又跑偏得比较严重，我觉得可能有些朋友看到那样的答案会被误导，所以就上去回答了一下这个问题。

下面我把问题和我的回答发布在这里，也供更多的人查看。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void) {
    int var;
    var = 88;
    if ((pid = vfork()) < 0) {
        printf("vfork error");
        exit(-1);
    } else if (pid == 0) { /* 子进程 */
        var++;
        return 0;
    }
    printf("pid=%d, glob=%d, var=%d\n", getpid(), glob, var);
    return 0;
}
```

基础知识

首先说一下fork和vfork的差别：

- fork 是 创建一个子进程，并把父进程的内存数据copy到子进程中。
- vfork是 创建一个子进程，并和父进程的内存数据share一起用。 这两个的差别是，一个是copy，一个是share。（关于fork，可以参看酷壳之前的《一道fork的面试题》）

你 man vfork 一下，你可以看到，vfork是这样的工作的，

1. 保证子进程先执行。
2. 当子进程调用exit()或exec()后，父进程往下执行。

那么，为什么要干出一个vfork这个玩意？原因在man page也讲得很清楚了：

> Historic Description
>
> Under Linux, fork(2) is implemented using copy-on-write pages, so the only penalty incurred by fork(2) is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child. However, in the bad old days a fork(2) would require making a complete copy of the caller's data space, often needlessly, since usually immediately afterwards an exec(3) is done. Thus, for greater efficiency, BSD introduced the vfork() system call, which did not fully copy the address space of the parent process, but borrowed the parent's memory and thread of control until a call to execve(2) or an exit occurred. The parent process was suspended while the child was using its resources. The use of vfork() was tricky: for example, not modifying data in the parent process depended on knowing which variables are held in a register.

意思是这样的—— 起初只有fork，但是很多程序在fork一个子进程后就exec一个外部程序，于是fork需要copy父进程的数据这个动作就变得毫无意了，而且这样干还很重（注：后来，fork做了优化，详见本文后面），所以，BSD搞出了个父子进程共享的 vfork，这样成本比较低。因此，vfork本就是为了exec而生。

为什么**return**会挂掉，**exit()**不会？

从上面我们知道，结束子进程的调用是exit()而不是return，如果你在vfork中return了，那么，这就意味main()函数return了，注意因为函数栈父子进程共享，所以整个程序的栈就跪了。

如果你在子进程中return，那么基本是下面的过程：

1. 子进程的main() 函数 return了，于是程序的函数栈发生了变化。
2. 而main()函数return后，通常会调用 exit()或相似的函数（如：_exit()，exitgroup()）
3. 这时，父进程收到子进程exit()，开始从vfork返回，但是尼玛，老子的栈都被你子进程给return干废掉了，你让我怎么执行？（注：栈会返回一个诡异一个栈地址，对于某些内核版本的实现，直接报"栈错误"就给跪了，然而，对于某些内核版本的实现，于是有可能会再次调用main()，于是进入了一个无限循环的结果，直到vfork 调用返回 error）

好了，现在再回到 return 和 exit，return会释放局部变量，并弹栈，回到上级函数执行。exit直接退掉。如果你用c++ 你就知道，return会调用局部对象的析构函数，exit不会。（注：exit不是系统调用，是glibc对系统调用 _exit()或_exitgroup()的封装）

可见，子进程调用exit() 没有修改函数栈，所以，父进程得以顺利执行。

关于**fork**的优化

很明显，fork太重，而vfork又太危险，所以，就有人开始优化fork这个系统调用。优化的技术用到了著名的写时拷贝（COW）。

也就是说，对于fork后并不是马上拷贝内存，而是只有你在需要改变的时候，才会从父进程中拷贝到子进程中，这样fork后立马执行exec的成本就非常小了。所以，Linux的Man Page中并不鼓励使用vfork() ——

> “ It is rather unfortunate that Linux revived this specter from the past. The BSD man page states: “This system call will be eliminated when proper system sharing mechanisms are implemented. Users should not depend on the memory sharing semantics of vfork() as it will, in that case, be made synonymous to fork(2).””

于是，从BSD4.4开始，他们让vfork和fork变成一样的了

但在后来，NetBSD 1.3 又把传统的vfork给捡了回来，说是vfork的性能在 Pentium Pro 200MHz 的机器（这机器好古董啊）上有可以提高几秒钟的性能。详情见——“NetBSD Documentation: Why implement traditional vfork()”

今天的Linux下，fork和vfork还是各是各的，不过，还是建议你不要用vfork，除非你非常关注性能。

（全文完）